Science of Computer Programming 78 (2013) 218–247

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Combining traits with boxes and ownership types in a Java-like setting☆

Lorenzo Bettini [a], Ferruccio Damiani [a,*], Kathrin Geilmann [b], Jan Schäfer [b]

[a] *Dipartimento di Informatica, Università di Torino, Italy*

[b] *Department of Computer Science, University of Kaiserslautern, Germany*

ARTICLE INFO

Article history:
Received 1 December 2010
Received in revised form 11 October 2011
Accepted 13 October 2011
Available online 21 October 2011

Keywords:
Boxes
Featherweight Java
Ownership types
Traits

ABSTRACT

The box model is a lightweight component model for the object-oriented paradigm, which structures the flat object-heap into hierarchical runtime components called *boxes*. Boxes have clear runtime boundaries that divide the objects of a box into objects that can be used to interact with the box (the *boundary* objects) and objects that are encapsulated and represent the state of the box (the *local* objects). The distinction into local and boundary objects is statically achieved by an *ownership type system* for boxes that uses domain annotations to classify objects into local and boundary objects and that guarantees that local objects can never be directly accessed by the context of a box. A *trait* is a set of methods divorced from any class hierarchy. Traits are units of fine-grained reuse that can be composed together to form classes or other traits. This paper integrates traits into an ownership type system for boxes. This combination is fruitful in two ways: it can statically guarantee encapsulation of objects and still provide fine-grained reuse among classes that goes beyond the possibilities of standard inheritance. It also solves a specific problem of the box ownership type system: namely that box classes cannot inherit from standard classes (and vice versa), and thus code sharing between these two kinds of classes was not possible in this setting so far. We present an ownership type system and the corresponding soundness proofs that guarantee encapsulation of objects in an object-oriented language with traits.

1. Introduction

Component-based software systems are built by linking components together or by building new components from existing ones. A key issue in component-based development is to have clear encapsulation boundaries for components. Knowing the component's boundaries enables modular analysis and the reuse of components in different program contexts without being bothered by unexpected interference between the component and its context. Most mainstream programming languages are object-oriented, based on the concept of classes, and do not have a language-level component concept other than single objects. Unfortunately, the granularity level of classes and objects is not appropriate for reuse. Since classes play the primary role of generators of instances, they must provide a complete set of features describing an object. Therefore, classes are often too coarse-grained to provide a minimal set of sensible reusable features [30]. Furthermore, in practice, objects usually rely on other objects to hold additional state and realize additional behavior, forming implicit components at runtime. Therefore, classes are often too fine-grained to provide a software fragment that can be specified,

☆ The authors of this paper are listed in alphabetical order. This work has been partially supported by the EU project FP7-231620 HATS, by the German–Italian University Centre (Vigoni program), and by MIUR (PRIN 2008 DISCO).

* Corresponding author. Tel.: +39 011 6706719; fax: +39 011 751603.
E-mail address: damiani@di.unito.it (F. Damiani).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.10.006

analyzed, and deployed independently [65]. Moreover, the two-level style provided by standard object-oriented languages (i.e., classes and objects) is known to be a limitation in some programming scenarios [38].

The box model [56,55,57] is a component model that defines components both on a syntactic level (by a set of classes) and as a runtime entity (as a set of objects). Components are instantiated and are similar to objects: they have an identity and a local state. A component instance is called a *box*. In general, a box may contain several objects and it can have nested boxes. To guarantee encapsulation of certain objects of a box an ownership type system [24,16,45,26] is used. It defines two ownership domains [3,61,55] for each box, namely a *local* and a *boundary* domain. Each object of a box and each inner box is located in one of these domains. Objects (and inner boxes) in the local domain are considered to be private to the component and are only accessible by objects of the same box and of inner boxes, whereas objects (boxes) in the boundary domain can be accessed by objects of other boxes. This is called the encapsulation property of the type system. Despite the restriction to only two domains, the box model is almost as expressive as the ownership domains approach, i.e., it can express design patterns like iterators but not like factories. Note that boxes are a runtime concept; thus, they do not aim at modeling static mechanisms such as *packages* or *namespaces*.

The notion of *trait* was introduced by Ungar et al. [66] in the dynamically typed prototype-based language Self to refer to a parent object to which an object may delegate some of its behavior. Subsequently, Schärli et al. [62,30] formulated and implemented traits in the dynamically typed class-based language Squeak/Smalltalk [11]. Traits have been designed to play the role of *units for fine-grained reuse* in order to counter the problems of class-based inheritance with respect to code reuse. A trait is a set of methods, completely independent from any class hierarchy. The common behavior (i.e., the common methods) of a set of classes can be factored into a trait. Various formulations of traits in a Java-like setting can be found in the literature (see, e.g., [63,52,12,60,13,43,42]). The recent programming language Fortress [6] (where there is no class-based inheritance) incorporates a form of the trait construct, while the "trait" construct incorporated in Scala [54] is indeed a form of mixin (mixins are subclasses parameterized over their superclasses; see, e.g., [18,41,33,7]).

The object-oriented calculi supporting the box model that have been proposed [56,55] do not model class-based inheritance. These calculi consider the following.

- *Interfaces*, as object types, defining only method signatures.
- *Box interfaces*, as box types, defining only method signatures.
- *Classes*, as generators of objects, implementing interfaces by defining (fields and) methods.
- *Box classes*, as generators of boxes, implementing box interfaces by defining (fields and) methods.
- *Ownership annotations*, to guarantee at compile time the encapsulation of runtime components.

Since the box ownership type system separates box and non-box type hierarchies, having a box class that is a subtype of a non-box class (or vice versa) would be rejected by the box ownership type system. Therefore, in order to add to these calculi a form of class-based inheritance that (as in mainstream programming languages like Java and C♯) identifies inheritance with subtyping, it is necessary to consider two separate class hierarchies, one for box classes and one for classes, that cannot share code through class-based inheritance.

In this paper, we exploit traits to support code reuse among box classes and classes. We present a Java-like minimal core calculus for boxes and traits. The calculus supports interface-based polymorphism, uses traits as units of fine-grained reuse that makes it possible to share code among box classes and classes, and is equipped with an ownership type system that supports encapsulation of runtime components. Namely, we consider: *Interfaces*, *Box interfaces* and *Ownership annotations* as above, and the following.

- *Traits*, as units of behavior reuse, defining only methods.
- *Classes*, as generators of objects, implementing interfaces (by defining fields and) by using traits.
- *Box classes*, as generators of boxes, implementing box interfaces (by defining fields and) by using traits.

In our proposal, traits are not types (that is, a trait declaration does not introduce a type). Hence, a class is not subtype of its composing traits, and the same trait can be used to compose both box classes and classes, without breaking the type system.

A desirable feature of a programming language with traits is the ability to analyze each trait definition in isolation from the classes and the traits that use it (see, e.g., [63]), thus avoiding reanalyzing a trait whenever it is used by a different class. As pointed out in [9,13], the fact that traits are not types makes it possible to include trait composition operations (like method exclusion and renaming) that do not preserve structural subtyping, thus increasing their potential for reuse. A constraint-based type system supporting these features in the context of a nominal Java-like type system has been proposed [13]. In the context of a programming language with boxes and ownership types, type-checking a trait in isolation from the classes that use it poses additional problems since, while type-checking the body of the method `m`, in order to be able to perform the ownership type-checks it is needed to know whether the class `C` of the `this` object is a box class. In this paper, we address this problem by showing that the constraint-based typing approach illustrated in [13] can be smoothly extended to deal with ownership types. The idea is to analyze the methods provided by a trait definition by using ownership type-constraints to collect the ownership type-checks that require to know the class `C` that contains the methods. These constraints will then be checked when type-checking the classes that use the trait.

To the best of our knowledge, this is the first attempt to define an ownership type system for a language with traits and the first attempt to combine boxes and traits. In order to focus on the interactions between traits and boxes and on the

interactions between traits and ownership types, we do not consider class-based inheritance in our calculus (along the lines and design choices of [12,10] and of the FORTRESS language).

Controlling the access to specific objects in specific contexts (boxes) is crucial for coordinating the access to sensitive data and for keeping consistency in an application. Our approach scales to a concurrent and distributed setting, since the synchronization of the access to data is orthogonal to our ownership type system. This type system may complement concurrency mechanisms providing guarantees that the access to specific resources is allowed only to the desired components.

A preliminary version of the results presented in this paper has been presented in [8]. This paper presents a slightly simplified version of the calculus, contains a new example, provides the complete formalization of the constraint-based ownership type system and of the operational semantics, and proves the soundness of the constraint-based ownership type system.

Organization of the paper. In Section 2, we introduce and motivate our proposal by an example. In Section 3, we present the syntax of the calculus. The ownership type system is presented in Sections 4–6. The operational semantics and the type and ownership soundness are presented in Sections 7 and 8, respectively. Related work is discussed in Section 9. We conclude by summarizing the paper and outlining possible directions for future work. The appendices contain the proofs of the main results.

2. Background and motivation

In Section 2.1, we briefly recall the programming model of boxes with ownership annotations in a standard JAVA-like setting with single inheritance. Then, in Section 2.2, we illustrate the combination of the box model with a trait-based object-oriented language and, in Section 2.3, we discuss the resulting benefits. As a running example, we use the implementation of a bank that manages an arbitrary number of bank accounts. The example aims at illustrating our proposal, rather than at providing a realistic case study.

2.1. Programming with boxes and ownership annotations

The box model extends the object-oriented programming world of interfaces, classes, objects, references, object-local state, and methods with components, which we call boxes. Similar to an object, a box is a runtime entity, which is created dynamically, and it has an identity and a state. In general, it groups several objects together, and its state is composed of the contained object states. At runtime, each object belongs to exactly one box, thus defining a clear runtime boundary. A more detailed description including the discussion of design decisions and showing the use of the model for modular specification can be found in [56].

On the source level, boxes are described by *box interfaces* and *box classes*. Each box class implements a box interface. When a box class is instantiated, a new box is created together with the object of the box class. The resulting object has the type of the corresponding box interface because (in order to enforce interface-based polymorphism) in our language the programmer can only use interfaces as types, not classes. Boxes form a tree at runtime, with a special global box at the root. A box is nested in the box that created it. The main expression of the program is always evaluated in the global box.

The purpose of the box model is to define a precise boundary of object-oriented runtime components. In addition, the box model conceptually structures the heap into hierarchical components. One important aspect of components is encapsulation. To ensure that certain objects are never exposed by a box, the object-oriented calculi supporting the box model that have been proposed [56,55] are equipped with an ownership type system. The basic idea is to group the objects of a box into distinct domains — a *local* and a *boundary* domain. Local objects are encapsulated in the box and cannot be referenced from the outside; boundary objects are accessible from the outside. The owner object of a box, i.e., the instance of the box class, is always accessible by the outside. It does not belong to the boundary domain of the box; instead, it belongs to some domain of its surrounding box. In general, the accessibility among objects follows three rules, called the *accessibility invariant*: (i) objects in the same box can access each other, (ii) when an object can access the owner object of a box, it can access the boundary objects of this box, and (iii) objects can access any object of a surrounding box (transitively).

This leads to a generalization of the *owners-as-dominators* property known from other ownership type systems [24], which we call *boundaries-as-dominators*. This property essentially means that all access paths from the environment of a box to a local object must go through the *boundary* of the box, where the boundary consists of the box owner, the objects of the boundary domain, and transitively the boundary of all boxes whose owners are in the boundary. The more restricted owners-as-dominators property can be achieved by a box without any boundary objects.

The ownership type system, which guarantees the accessibility invariant, relies on source level type annotations. A type `I` is annotated with a domain annotation `d` by writing `d I`. A domain annotation can be `local`, `boundary`, `owner`, or `global`. In addition, types can have domain parameters to express genericity in domain annotations. A domain annotation is always relative to a certain box. By default, this is the current box, i.e., the box of the `this`-object. For example, the type `local I` means that all instances of that type belong to the local domain of the current box. A box can also be explicitly specified by using a local variable referencing a box owner, i.e., an object of a box class. For example, the domain annotation

```
box interface IBank {
    global String getAddress();
    boundary IAccess getAccess(int accountid);
}

interface IAccess {
    global String getAddress();
    int getBalance(int pin);
    void transferTo(boundary IAccess acc, int amount, int pin);
}
```

```
interface IAccount {
    global String getAddress();
    void withdraw(int amount);
    void deposit(int amount);
    int getBalance();
    boolean checkPin(int pin);
    boundary IAccess getAccess();
}
```

Fig. 1. Interfaces of the bank example.

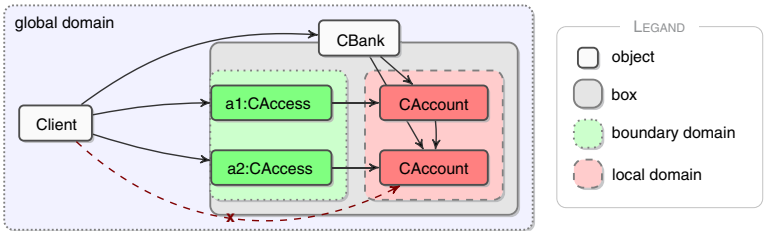

Fig. 2. Runtime view to the bank example. The box ownership type system ensures that the client can access the boundary objects of the bank, i.e., the `CAccess` objects, but cannot access local objects of the bank, i.e., the `CAccount` objects (indicated by a crossed-out line). Thus the bank implementation has the guarantee that internal account objects are encapsulated and cannot be directly manipulated by clients.

`x.boundary` refers to the boundary domain of the box of the object referred by `x`.[1] The `owner` domain annotation refers to the domain which the current `this`-object belongs to, and the `global` domain annotation represents the local domain of the global box. Types are only assignable if they have compatible domain annotations. Domain annotations are compatible if at runtime they always refer to the same domain. The domain annotations restrict the usage of types to guarantee the encapsulation of objects. In particular, it is guaranteed that all access paths from an object outside of a box to an object of a local domain of a box must go through either the owner of the box or a boundary object. This is ensured by the type system by the restricting of assignments as described above and the prevention of certain domain annotations like `x.local`, for example.

2.1.1. The bank example using boxes and ownership annotations

The interfaces of the implementation are shown in Fig. 1. The basic idea of the implementation is that a bank represents internally accounts as `IAccount` objects and that external access to these object is only done via `IAccess` objects. Fig. 2 shows a runtime view of the bank box with two accounts and a single client that accesses these two accounts via access objects, incorporating the concepts of boxes and domains.

As the bank implementation requires encapsulating objects, it is realized as a box; thus the `IBank` interface is declared as a `box` interface. The bank has an address which can be obtained by the `getAddress` method, which returns a `String` object. As the `String` object is immutable, it can be safely shared, and thus it is put in the `global` domain. In order to be able to access an account of the bank, a client has to obtain an `IAccess` object by using the `getAccess` method of the bank. As the `IAccess` objects should be accessible by clients of the bank, they are put into the `boundary` domain of the bank box. Internally a bank uses `IAccount` objects to manage bank accounts. These objects, however, must not be accessible by clients directly as they offer operations that can arbitrarily manipulate the state of an account. For simplicity, our example only allows for bank-internal transfer of money, i.e., money can only be transferred between accounts of the same bank. To transfer money, a client uses the `IAccount` interface and passes as argument the access object of the account to which the money should be transferred too. The `boundary` annotation ensures that the access object belongs to the same bank as the receiver object. Fig. 3 shows two client implementations that demonstrate this.

The implementation of the bank in terms of box classes and classes with standard class-based inheritance is shown in Fig. 4. Internal account implementations are realized by the `CAccount` class. The `CBank` class has a `local Map` that maps account identifiers to `local CAccount` objects. The `local` ownership annotation ensures that these objects can only be referenced by objects that belong to the same bank box. The `CAccount` class offers a method `getAccess` that returns a new `boundary IAccess` object to access the account. As `CAccount` is not a box class itself, the `boundary` annotation refers to the boundary domain of the owning box: in this example, a bank box. The `CAccess` class implements the methods to access the internal account objects. It has a `local IAccount` field that refers to the corresponding account object. The `local` annotation ensures that the account object belongs to the local domain of the bank box.

[1] To simplify the semantics of our language, all local variables are `final` and can thus be used as box owners. In a language like Java, non-`final` variables cannot be used as owners. This is similar to other ownership domain approaches [3,55,61].

```
class GoodClient implements IClient {
  void useBank(final IBank bank) {
    bank.boundary IAccess a1 = bank.getAccess(1001);
    bank.boundary IAccess a2 = bank.getAccess(1002);
    a1.transfer(a2, 100, 1234); // type correct
  }
}
```

```
class MaliciousClient implements IClient {
  void useBank(final IBank bank) {
    final IBank bbank = new BadBank();
    bank.boundary IAccess goodA = bank.getAccess(1001);
    bbank.boundary IAccess badA = bbank.getAccess(1002);
    badA.transfer(goodA, −100, 1234); // type error
  }
}
```

Fig. 3. Two bank clients that try to transfer money between two accounts. The `GoodClient` uses two access objects of the same bank and can thus successfully transfer money. The `MaliciousClient` tries to use access objects of different banks and will be rejected by our box ownership type system.

```
class CAccount implements IAccount {
    int balance;
    int pin;
    global String address;
    CAccount(int aPin, global String anAddress)
      { ... /* init fields */}
    global String getAddress() { return address;}
    boolean checkPin(int aPin) { return pin == aPin; }
    void withdraw(int amount) { balance −= amount; }
    void deposit(int amount) { balance += amount; }
    int getBalance() { return balance; }
    boundary IAccess getAccess() { return new boundary CAccess(this); }
}

class CAccess implements IAccess {
    local IAccount acc;
    CAccess(local IAccount aAcc)
      { acc = aAcc; }
    global String getAddress() { return acc.getAddress();}
    int getBalance(int pin) {
      if (!acc.checkPin(pin)) fail;
      return acc.getBalance();
    }
    void transferTo(boundary IAccess toAcc, int amount, int pin) {
      if (!acc.checkPin(pin)) fail;
      acc.withdraw(amount);
      toAcc.deposit(amount);
    }
}
```

```
box class CBank implements IBank {
    local Map<Integer, local IAccount> accounts;
    global String bankAddress;
    CBank(global String address)
      { ... /* init fields */}
    global String getAddress() {return bankAddress;}
    boundary IAccess getAccess(int accountid)
      { return accounts.get(accountid).getAccess(); }
}

class CFeeAccount extends CAccount {
  int fee;
  CFeeAccount(int aPin, global String anAddress, int aFee)
    { ... /* init fields */}
  void withdraw(int amount) { balance −= amount + fee; }
}

class CAuthAccount extends CAccount {
  CAuthAccount(int aPin, global String anAddress)
    { ... /*init fields*/}
  boolean checkPin(int aPin) {
    ... // authentication method
  }
}

class CFeeAuthAccount extends CFeeAccount {
  CFeeAuthAccount(int aPin, global String anAddress, int aFee)
    { ... /* init fields */}
  boolean checkPin(int aPin) {
    ... // same code as in CAuthAccount
  }
}
```

Fig. 4. Implementation of the bank example that uses class-based inheritance and the box ownership type system to ensure state encapsulation.

Besides the standard `CAccount` implementation, the bank also provides accounts with additional authentication and billing functionality (`CFeeAccount` and `CAuthAccount`). Both classes inherit from the `CAccount` class to share code with the basic account implementation. However, an implementation of an account class that has both functionalities (`CFeeAuthAccount`) cannot be realized easily in a language with single inheritance without duplicating code.

2.2. Programming with boxes, ownership annotations, and traits

In this section, we illustrate how traits can be exploited to support code reuse among box classes and classes, and also to improve code reuse among box classes and among classes. We consider a language where a trait consists of *methods*, of *required methods*, which parameterize the behavior, and of *required fields* that can be directly accessed in the body of the methods, along the lines of [9,13]. Traits are building blocks to compose classes and other traits. A suite of trait composition operations allows the programmer to build classes and composite traits.

In the languages considered in this paper, a trait is not a type. That is, a trait declaration does not introduce a type. Hence a class is not subtype of its composing traits, and the same trait can be used to compose both box classes and classes. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state. Therefore a class composed by using traits has to provide the required fields.

The trait composition operations considered in our language are as follows: a *basic trait* defines a set of methods and declares the required fields and the required methods. The *symmetric sum* operation, $+$, merges two traits to form a new trait. It requires that the summed traits are disjoint. That is, they do not define identically named methods, they have compatible requirements (two requirements on the same method/field name are compatible if they are identical, while requirements

```
trait TAddress {
  global String address;
  global String getAddress() {return address;}
}

trait TAccount is TAddress + {
    int balance; int pin;
    boolean checkPin(int aPin) { return pin == aPin; }
    void withdraw(int amount) { balance −= amount; }
    void deposit(int amount) { balance += amount; }
    int getBalance() { return balance; }
    boundary IAccess getAccess() { return new boundary CAccess(this); }
}

class CAccount implements IAccount by TAccount {
  int balance;
  int pin;
  global String address;
  CAccount(int aPin, global String anAddress)
    { ... /* init fields */}
}

trait TAccess {
    local IAccount acc;
    global String getAddress() {return acc.getAddress();}
    int getBalance(int pin) {
        if (!acc.checkPin(pin)) fail;
        return acc.getBalance();
    }
    void transferTo(boundary IAccess toAcc, int amount, int pin) {
        if (!acc.checkPin(pin)) fail;
        acc.withdraw(amount);
        toAcc.deposit(amount);
    }
}

class CAccess implements IAccess by TAccess {
  local IAccount acc;
  CAccess(local IAccount anAcc)
    { this.acc = anAcc; }
}
```

```
trait TBank is TAddress + {
    local Map<Integer, local IAccount> accounts;
    boundary IAccess getAccess(int accountid)
      { return accounts.get(accountid).getAccess(); }
}

box class CBank implements IBank
    by TBank[address renameTo bankAddress] {
    local Map<Integer, local IAccount> accounts;
    global String bankAddress;
    CBank(global String address)
      { ... /* init fields */}
}

trait TFee {
  int fee;
  void withdraw(int amount) { balance −= amount + fee; }
}

class CFeeAccount implements IAccount
  by TAccount[exclude withdraw] + TFee {
  int balance;
  int pin;
  int fee;
  global String address;
  CFeeAccount(int aPin, global String anAddress, int aFee)
    { ... /* init fields */}
}

trait TAuth {
  boolean checkPin(int aPin) {
    ... // authentication method
  }
}

class CAuthAccount implements IAccount
  by TAccount[exclude checkPin] + TAuth {
  int balance;
  int pin;
  global String address;
  CAuthAccount(int aPin, global String anAddress)
    { ... /* init fields */}
}

class CFeeAuthAccount implements IAccount
  by TAccount[exclude checkPin][exclude withdraw] + TAuth + TFee {
  int balance;
  int pin;
  int fee;
  global String address;
  CFeeAuthAccount(int aPin, global String anAddress, int aFee)
    { ... /* init fields */}
}
```

Fig. 5. Implementation of the bank example using boxes and traits.

on different method/field names are always compatible), and the methods defined by each trait are compatible with the methods required by the other trait. The operation `exclude` forms a new trait by removing a method from an existing trait. The operation `aliasAs` forms a new trait by adding a copy of an existing method with a new name. The original method is still available and, when a recursive method is aliased, its recursive invocation refers to the original method. The operation `renameTo` forms a new trait by renaming all the occurrences of a required field name or of a required/provided method name in an existing trait. In order to focus on the interactions between traits and boxes and on the interactions between traits and ownership types, we do not consider class-based inheritance in our calculus.

2.2.1. The bank example using boxes, ownership annotations, and traits

In Fig. 5, we show the implementation of the bank example using traits. Traits are declared with the keyword `trait` and a name followed by a trait expression. Trait expressions are defined by the trait composition operations described above.

In our example, all interfaces require the implementation of a `getAddress` method. We implement this method in the basic trait `TAddress`. This trait gets reused in the trait `TAccount` and `TBank`, and thus the method is available in all account classes and in the class `CBank`. Note that the account classes directly provide the required field, whereas the class `CBank` provides the field `bankAddress`. In order to match required and provided field, the required field of the trait is renamed during the composition of the traits into `CBank`. Using the trait `TAddress` directly or indirectly in multiple classes, which

are unrelated in terms of types, theses classes share the same implementation. In single-inheritance-based languages one would achieve this by using a common supertype, but at the cost of adding an unwanted subtype relation. Also note that the trait `TAccess`, which implements the methods of the interface `IAccess`, provides a different implementation of a `getAddress` method, and therefore the trait `TAddress` is not used to implement the class `CAccess`. We can see that using traits is independent of the type hierarchy formed by the interfaces.

The trait `TAccount` provides implementations for all methods of the `IAccount` interface, whereas the trait `TFee` only provides a new `withdraw` method. The class `CAccount`, representing standard accounts, is built by directly using the trait `TAccount` and declaring all fields required by the trait. Since classes (and not traits) are the generators of objects, constructors are implemented inside classes. All other methods are implemented by traits, which are given in the trait expression of the class declarations. The class `CFeeAccount` is defined by using the trait `TAccount` and the trait `TFee`. We exclude the implementation of `withdraw` from the `TAccount` trait and use the `withdraw` operation of `TFee` instead. This is similar to overriding in inheritance-based languages, but with traits we do not introduce additional subtype relations. The class `CAuthAccount` is implemented similarly to `CFeeAccount`.

In an object-oriented language with single inheritance, it would be difficult to create a fourth class that combines a `CFeeAccount` with a `CAuthAccount`. Typically, one would have to inherit one class and manually add the code of the other class to the subclass. When using traits as unit of code reuse, we can combine the traits `TFee` and `TAuth` with the trait `TAccount` into the class `CFeeAuthAccount` without the need to copy neither code for handling fees nor code for handling authentication. Instead we only have to remove the methods provided by `TFee` and `TAuth` from `TAccount`, in order to define a valid trait expression.

2.3. Advantages of combining boxes with traits

Up to now, the box model has been presented in class-based languages without inheritance [56,55]. The box model could be extended to languages with single inheritance, as done for other ownership type systems and illustrated in the example in Section 2.1.1. However, the box ownership type system separates box and non-box type hierarchies, which means that a box class cannot be a subtype of a non-box class and vice versa. As mainstream programming languages like Java and C♯ identify inheritance and subtyping, these two hierarchies cannot share code through class-based inheritance. For instance, in Fig. 4, the body of the method `getAddress` is duplicated in the box class `CBank` and in the class `CAccount`.

A possible way to deal with this problem might be to introduce a new type parameter that defines for each type (class or interface) use whether the type is a box type or not. This solution has the drawback that it would make the language and the type system more complex, both for the user and for the implementer.

As illustrated in Section 2.2, using traits elegantly solves this problem, as traits can be shared among classes even if one is a box class and the other is not. For instance, in Fig. 5, the method `getAddress` is defined in trait `TAddress` and then used by the class `CAccount` and by the box class `CBank`. Moreover, traits also improve code reuse among box classes and among classes. For instance, in Fig. 5, the method `checkPin` defined in trait `TAuth` is used by both classes `CAuthAccount` and `CFeeAuthAccount`.

3. A minimal core calculus for boxes and traits: syntax and flattening

In this section, we present the syntax of Imperative Featherweight Box Trait Java (IFBTJ), a minimal core language (in the spirit of FJ [36]) for boxes and traits. We also present a flattening translation that provides a canonical semantics of traits by compiling them away.

3.1. Syntax

The syntax of IFBTJ is presented in Fig. 6. We use similar notations as FJ [36]. For instance: $\overline{\mathtt{e}}$ denotes the possibly empty sequence $\mathtt{e}_1, \ldots, \mathtt{e}_n$ and the pair $\overline{\mathtt{N}}\ \overline{\mathtt{f}}$; stands for $\mathtt{N}_1\ \mathtt{f}_1; \ldots \mathtt{N}_n\ \mathtt{f}_n;$. The empty sequence is denoted by $\bullet$, the length of a sequence $\overline{\mathtt{e}}$ is denoted by $|\overline{\mathtt{e}}|$, and the concatenation of two sequences $\overline{\mathtt{N}}'$ and $\overline{\mathtt{N}}''$ is denoted by $\overline{\mathtt{N}}'\overline{\mathtt{N}}''$.

A program consists of interfaces, box interfaces, traits, classes, box classes, and an expression, which represents the main method of the program. For simplicity, we assume that each class (and each box class) has a companion interface (respectively, box interface) that it implements. Interfaces and box interfaces list the public methods of a class. The language has no explicit constructors: when a new object is created, all fields are set to `null`. Note that constructors can be simulated by ordinary method calls. Each class or box class declares fields and defines methods through a trait expression.

Interface names and box interface names are the only source level types. The set of expressions is quite standard. Just observe that, since interface names and box interface names are the only source level types, fields can be selected only on `this`.

For conciseness of the formalization, we have streamlined the notation of ownership annotations used in Section 2. Instead of writing the owning domain in front of the type, we now write it as the first parameter of the type. This also means that there is no `owner` keyword, because the obligatory first domain parameter always represents the owning domain. A domain annotation can either be a domain parameter α, the global domain, or is of the form `b.c`, where the first part defines

P	::=	$\overline{\texttt{ID}}\ \overline{\texttt{TD}}\ \overline{\texttt{CD}}\ \texttt{e}$	programs
ID	::=	[box] interface $\texttt{I}\langle\overline{\alpha}\rangle$ extends $\overline{\texttt{N}}$ { $\overline{\texttt{S}}$ }	interfaces
N	::=	$\texttt{I}\langle\overline{\texttt{d}}\rangle$	source types
S	::=	$\texttt{N}\ \texttt{m}\ (\overline{\texttt{N}}\ \overline{\texttt{x}})$	method headers
d	::=	α | b.c | global	domain annotations
b	::=	box | x | $\underline{\texttt{null}}$ | $\underline{\texttt{?}}$	domain owners
c	::=	local | boundary	domain kinds
TD	::=	$\texttt{trait}\ \texttt{T}\langle\overline{\alpha}\rangle\ \texttt{is}\ \texttt{TE}$	traits
TE	::=	$\{\overline{\texttt{F}};\ \overline{\texttt{S}};\ \overline{\texttt{M}}\}$ | $\texttt{T}\langle\overline{\texttt{d}}\rangle$ | TE + TE | TE[exclude m] | TE[m aliasAs m] | TE[m renameTo m] | TE[f renameTo f]	trait expressions
F	::=	N f	fields
M	::=	S { return e; }	methods
e	::=	x | null | this.f | this.f = e | e.m($\overline{\texttt{e}}$) | new $\texttt{C}\langle\overline{\texttt{d}}\rangle$ | (N)e |	expressions
CD	::=	[box] class $\texttt{C}\langle\overline{\alpha}\rangle$ implements $\texttt{I}\langle\overline{\texttt{d}}\rangle$ by TE { $\overline{\texttt{F}};$ }	classes

Fig. 6. IFBTJ: Syntax ($\texttt{I} \in$ interface names, $\texttt{T} \in$ trait names, $\texttt{C} \in$ class names, $\texttt{m} \in$ method names, $\texttt{f} \in$ field names, $\alpha, \beta \in$ domain parameters).

the owner of the domain, and the second part defines the domain kind, that is, whether it is the boundary or local domain. The keyword box denotes the owner of the current box. The name of a local variable x is used for objects of box classes and denotes the box owned by x. For example, x.local denotes the local domain of the box owned by x. In general, this can also be an owner of a domain, but we assume that in the surface syntax this does not appear as an owner. It may, however, appear as owner during the typing. Owners null and ? do not belong to the surface syntax (indicated by an underline), but can appear during reduction. ? as owner represents an invalid domain annotation, and null is the owner of the global domain. In fact, all occurrences of global are treated as null.local.

Convention 3.1 (*Conventions on Sequences of Named Elements*)**.** *We use the phrase* sequence of named elements *to refer to a sequence of declarations (e.g., field declarations, method headers, method definitions, and so on). Unless explicitly stated (or clear from the context), we do not consider a sequence of names as a sequence of named elements. We say that a sequence of named elements is* well formed *if it does not contain duplicated names. In the following, sequences of named elements are in general assumed to be well formed. Sometimes we emphasize this fact by writing* $\overline{\texttt{S}}$ **wf** *to assert that* $\overline{\texttt{S}}$ *is well-formed. The sequence of the element names of* $\overline{\texttt{S}}$ *is denoted by* $names(\overline{\texttt{S}})$*, the subsequence of the elements of* $\overline{\texttt{S}}$ *with names* $\overline{\texttt{n}}$ *is denoted by* $choose(\overline{\texttt{S}}, \overline{\texttt{n}})$*, and* $exclude(\overline{\texttt{S}}, \overline{\texttt{n}})$ *denotes the sequence obtained from* $\overline{\texttt{S}}$ *by removing the elements with names* $\overline{\texttt{n}}$*. Following [36], we use a set-based notation for operators over sequences of named elements. For instance,* $\texttt{M} = \texttt{N}\ \texttt{m}(\texttt{N}\ \texttt{x})\{\texttt{return e}\} \in \overline{\texttt{M}}$ *means that the method definition* M *occurs in* $\overline{\texttt{M}}$*. In the union and in the intersection of sequences, denoted by* $\overline{\texttt{S}} \cup \overline{\texttt{S}}'$ *and* $\overline{\texttt{S}} \cap \overline{\texttt{S}}'$*, respectively, it is assumed that if* $\texttt{n} \in names(\overline{\texttt{S}})$ *and* $\texttt{n} \in names(\overline{\texttt{S}}')$ *then* $choose(\overline{\texttt{S}}, \texttt{n}) = choose(\overline{\texttt{S}}', \texttt{n})$*. In the disjoint union of sequences, denoted by* $\overline{\texttt{S}} \cdot \overline{\texttt{S}}'$*, it is assumed that* $names(\overline{\texttt{S}}) \cap names(\overline{\texttt{S}}') = \varnothing$*.*

3.2. Flattening

The *flattening principle* has been introduced in the original formulation of traits in Squeak/Smalltalk [30] in order to provide a canonical semantics to traits. Flattening states that the semantics of a method introduced in a class through a trait should be identical to the semantics of the same method defined directly within a class. This makes it possible to reason about the properties of a language with traits by relying on the semantics of the subset of the language without traits. Note that flattening aims only to provide a canonical semantics to traits; it is not an especially effective implementation technique (see, e.g., [52,39]).

In order to formalize flattening for IFBTJ, we consider a subset of the language that we call FIFBTJ (Flat IFBTJ), where there are no trait declarations and the syntax of trait expressions is simplified as follows:

$$\texttt{TE} ::= \{\overline{\texttt{F}};\ \bullet;\ \overline{\texttt{M}}\}$$

A FIFBTJ class $\texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\overline{\texttt{d}}\rangle\ \texttt{by}\ \{\ \overline{\texttt{F}}';\ \bullet;\ \overline{\texttt{M}}\ \}\ \{\ \overline{\texttt{F}};\ \}$ can be understood (modulo the domain annotations) as the standard Java class $\texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\overline{\texttt{d}}\rangle\ \{\ \overline{\texttt{F}};\ \overline{\texttt{M}}\ \}$. Similarly for box classes. Therefore, the canonical (static and dynamic) semantics for IFBTJ can be specified by providing: (i) a semantics for FIFBTJ, and (ii) a flattening translation that maps a IFBTJ program into a FIFBTJ program.

The flattening translation is specified through the function $[\![\cdot]\!]$, given in Fig. 7, that maps each IFBTJ class or box class declaration to a FIFBTJ class or box class declaration, respectively, and maps a trait expression to a sequence of method declarations. We write $[\![\texttt{P}]\!]$ to denote the program obtained from P by dropping all the trait declarations and by translating

$$
\begin{array}{rcl}
[\![\texttt{[box] class C}\langle\overline{\alpha}\rangle\ \texttt{implements I}\langle\overline{\texttt{d}}\rangle\ \texttt{by TE}\ \{\ \overline{\texttt{F}};\ \}]\!] & \stackrel{\text{def}}{=} & \texttt{[box] class C}\langle\overline{\alpha}\rangle\ \texttt{implements I}\langle\overline{\texttt{d}}\rangle\ \texttt{by}\ \{\ \overline{\texttt{F}};\ \bullet;\ [\![\texttt{TE}]\!]\ \}\ \{\ \overline{\texttt{F}};\ \} \\
[\![\{\ \overline{\texttt{F}};\ \overline{\texttt{S}};\ \overline{\texttt{M}}\ \}]\!] & \stackrel{\text{def}}{=} & \overline{\texttt{M}} \\
[\![\texttt{T}\langle\overline{\texttt{d}}\rangle]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}[\overline{\texttt{d}}/\overline{\alpha}]]\!] \quad \textbf{if}\ \texttt{trait T}\langle\overline{\alpha}\rangle\ \texttt{is TE} \\
[\![\texttt{TE}_1 + \texttt{TE}_2]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}_1]\!] \cdot [\![\texttt{TE}_2]\!] \\
[\![\texttt{TE [exclude m]}]\!] & \stackrel{\text{def}}{=} & \overline{\texttt{M}}' \cdot \overline{\texttt{M}}'' \quad \textbf{if}\ [\![\texttt{TE}]\!] = \overline{\texttt{M}}' \cdot \texttt{M} \cdot \overline{\texttt{M}}''\ \textbf{and}\ \texttt{M} = \cdots\texttt{m}(\cdots)\{\cdots\} \\
[\![\texttt{TE [m aliasAs m']}]\!] & \stackrel{\text{def}}{=} & \overline{\texttt{M}} \cdot (\texttt{N m}'(\overline{\texttt{N}}\ \overline{\texttt{x}})\{\texttt{return e; }\}) \\
 & & \quad \textbf{if}\ [\![\texttt{TE}]\!] = \overline{\texttt{M}}\ \textbf{and}\ \texttt{N m}(\overline{\texttt{N}}\ \overline{\texttt{x}})\{\texttt{return e; }\} \in \overline{\texttt{M}} \\
[\![\texttt{TE[f renameTo f']}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{TE}]\!][\texttt{f}'/\texttt{f}] \\
[\![\texttt{TE[m renameTo m']}]\!] & \stackrel{\text{def}}{=} & \mathit{mR}([\![\texttt{TE}]\!], \texttt{m}, \texttt{m}') \\
\mathit{mR}(\texttt{N n}(\overline{\texttt{N}}\ \overline{\texttt{x}})\{\texttt{return e; }\}, \texttt{m}, \texttt{m}') & \stackrel{\text{def}}{=} & \texttt{N n}[\texttt{m}'/\texttt{m}](\overline{\texttt{N}}\ \overline{\texttt{x}})\{\texttt{return e}[\texttt{this.m}'/\texttt{this.m}]; \} \\
\mathit{mR}(\texttt{M}_1 \cdot \ldots \cdot \texttt{M}_n, \texttt{m}, \texttt{m}') & \stackrel{\text{def}}{=} & (\mathit{mR}(\texttt{M}_1, \texttt{m}, \texttt{m}')) \cdot \ldots \cdot (\mathit{mR}(\texttt{M}_n, \texttt{m}, \texttt{m}'))
\end{array}
$$

Fig. 7. Flattening IFBTJ to FIFBTJ.

$$
\begin{array}{lcll}
\texttt{U} & ::= & \texttt{I} \mid \texttt{C} & \text{type names} \\
\texttt{L} & ::= & \texttt{G} \mid \bot & \text{types} \\
\texttt{G} & ::= & \texttt{N} \mid \texttt{C}\langle\overline{\texttt{d}}\rangle & \text{nominal types}
\end{array}
$$

Fig. 8. IFBTJ: Type names, types and nominal types.

all the class and box class declarations. The clauses in Fig. 7 should be self-explanatory. Note that the flattening clause for field renaming is simpler than the flattening clause for method renaming (which uses the auxiliary function *mR*); this is due to the fact that fields can be accessed only on `this`.

4. Ownership typing for FIFBTJ

Before introducing the constraint-based ownership type system for IFBTJ, we introduce an ownership type system for FIFBTJ. The ownership type system for FIFBTJ is similar to the ownership type system described in [55]. This type system serves as a "specification" for the constraint-based ownership type system that we will present in Section 5, in the sense that any FIFBTJ program must be typable by the constraint-based ownership type system if and only if it is typable by the ownership type system. This property will be formally stated in Section 6.

In presenting the ownership type system, we will use the distinguished symbol $\bot$ to denote the type for the constant `null`, use the metavariable `U` to range over interface names and class names, and use the metavariables `L` and `G` to range over types and non-$\bot$ types, respectively; see Fig. 8.

Type environments, ranged over by Γ, are maps from variables to types. A type environment records the type information of free variables.

$$\Gamma ::= \texttt{this} : \texttt{G} \mid \Gamma, \texttt{x} : \texttt{N} \qquad \text{environment}$$

When building type environments with sequences of variables, e.g. $\overline{\texttt{x}} : \overline{\texttt{N}}$, we implicitly assume that the names $\overline{x}$ are distinct and do not contain `this`. We also use the functional notation $\Gamma(\texttt{x})$ to get the type of `x` in Γ.

In the following subsections, the different judgments are presented. They are shown in Fig. 9. Some judgments are of the form $\Gamma \vdash_{\circ} \ldots$. This means that the right-hand side is evaluated under the type environment Γ. The type of `this` plays a crucial role as it defines the valid domain parameters and whether the current context is a box class or a normal class. Therefore, a type assumption for `this` is required even when typing expression that do not contain occurrences of `this`.

4.1. Method and field lookup

Fig. 10 presents functions to look up field and method information from classes and interfaces. All lookup functions take the current context type as a parameter and substitute domain arguments for domain parameters. The function *field* looks up the type of a field in a class. The function *mbody* looks up the *body* of a method in a class, which is a pair $(\overline{x}, \texttt{e})$ of the parameter names $\overline{x}$ and the body expression `e` of the corresponding method. Finally, function *mSig* looks up method *signatures*. Signatures, ranged over by σ, are method headers deprived of parameter names. For instance, the signature associated to the header $\texttt{N m}(\texttt{N}_1\ \texttt{x}_1, \ldots, \texttt{N}_n\ \texttt{x}_n)$ is $\texttt{N m}(\texttt{N}_1, \ldots, \texttt{N}_n)$. The function *mSig* uses the function *toSig*, which converts method headers and methods into method signatures.

Function *params* returns the domain parameters of a class or an interface.

$\Gamma \vdash_o \diamond$	environment Γ is valid
$\Gamma \vdash_o \mathtt{d}_1 \rightarrow \mathtt{d}_2$	domain $\mathtt{d}_1$ can access domain $\mathtt{d}_2$
$\Gamma \vdash_o \mathtt{d}$	domain $\mathtt{d}$ is valid
$\Gamma \vdash_o \mathtt{G}'$	type $\mathtt{G}'$ is valid
$\Gamma \vdash_o \mathtt{e} : \mathtt{G}'$	expression $\mathtt{e}$ has type $\mathtt{G}'$
$\mathtt{G} \vdash_o \mathtt{S}$	method header $\mathtt{S}$ is well-typed
$\mathtt{G} \vdash_o \mathtt{M}$	method $\mathtt{M}$ is well-typed
$\mathtt{G} <: \mathtt{G}'$	type $\mathtt{G}$ is a subtype of type $\mathtt{G}'$
$\vdash_o \mathtt{ID}$	interface $\mathtt{ID}$ is well-typed
$\vdash_o \mathtt{CD}$	class $\mathtt{CD}$ is well-typed
$\vdash_o \mathtt{P} : \mathtt{N}$	program $\mathtt{P}$ is well-typed and its main expression has type $\mathtt{N}$

Fig. 9. Judgments used by the type system.

$$\frac{\ldots\ \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \ldots\ \mathtt{N}\ f;\ \ldots}{\mathit{field}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle, f) = \mathtt{N}\,[\overline{\mathtt{d}}/\overline{\alpha}]} \qquad \frac{\ldots\ \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \ldots\ \mathtt{N}\ \mathtt{m}(\overline{\mathtt{N}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e}\}\ \ldots}{\mathit{mbody}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle, \mathtt{m}) = (\overline{\mathtt{x}}, \mathtt{e}[\overline{\mathtt{d}}/\overline{\alpha}])} \qquad \frac{}{\mathit{toSig}(\mathtt{N}\ \mathtt{m}(\overline{\mathtt{N}}\ \overline{\mathtt{x}})) = \mathtt{N}\ \mathtt{m}(\overline{\mathtt{N}})}$$

$$\frac{}{\mathit{toSig}(\mathtt{S}\ \{\mathtt{return}\ \mathtt{e}\}) = \mathit{toSig}(\mathtt{S})} \qquad \frac{\ldots\ \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{N}\ \{\ \overline{\mathtt{F}};\ \overline{\mathtt{M}}\ \} \qquad \overline{\sigma} = \mathit{toSig}(\mathtt{M}_1) \cup \ldots \cup \mathit{toSig}(\mathtt{M}_n)}{\mathit{mSig}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle) = \overline{\sigma}\,[\overline{\mathtt{d}}/\overline{\alpha}]}$$

$$\frac{\ldots\ \mathtt{interface}\ \mathtt{I}\langle\overline{\alpha}\rangle\ \mathtt{extends}\ \overline{\mathtt{N}}\ \{\ \overline{\mathtt{S}}\ \} \qquad \overline{\sigma} = \mathit{mSig}(\mathtt{N}_1[\overline{\mathtt{d}}/\overline{\alpha}]) \cup \ldots \cup \mathit{mSig}(\mathtt{N}_n[\overline{\mathtt{d}}/\overline{\alpha}])}{\mathit{mSig}(\mathtt{I}\langle\overline{\mathtt{d}}\rangle) = (\mathit{toSig}(\overline{\mathtt{S}})\,[\overline{\mathtt{d}}/\overline{\alpha}]) \cup \overline{\sigma}} \qquad \frac{\mathtt{box}\ \mathtt{interface}\ \mathtt{I}\langle\overline{\alpha}\rangle\ \ldots}{\mathit{isBoxType}(\mathtt{I})}$$

$$\frac{\mathtt{box}\ \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \ldots}{\mathit{isBoxType}(\mathtt{C})} \qquad \frac{\mathit{isBoxType}(\mathtt{U})}{\mathit{isBoxType}(\mathtt{U}\langle\overline{\mathtt{d}}\rangle)} \qquad \frac{\ldots\ \mathtt{interface}\ \mathtt{I}\langle\overline{\alpha}\rangle\ \ldots}{\mathit{params}(\mathtt{I}) = \overline{\alpha}} \qquad \frac{\ldots\ \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \ldots}{\mathit{params}(\mathtt{C}) = \overline{\alpha}}$$

Fig. 10. Auxiliary functions to look up information from class and interface declarations.

4.2. Owning box

To be able to adapt the `box` keyword that is used as a domain owner to the using context, we define a function that translates the keyword to a matching representation. First, we define a function *obox* that, given a type and an expression, returns the box owner (see Fig. 11). The type and the expression represent the receiver of a method call or field access. Additionally it takes as the first parameter the owning domain of the context type, i.e., the domain which owns the type of the class or interface surrounding the call or field access. The expression is needed for cases where the receiver class is a box class as in that case the expression is used to represent the box in the calling context. Expressions, however, can only represent domain owners if they are either variables or `null`, specified by the *validOwner* function. In case the expression is not a valid domain owner, the invalid owner `?` is returned. If the type argument is not of a box type, the first domain parameter of the type is used to find out the box owner. If the domain parameter is α, we know that the owning box must be the same as the box we are currently in. If the type is not in the same box and no concrete domain is given as the owning domain, we cannot statically find out in which box the type argument is and therefore cannot return its owner. The actual translation of types to the user context is done by rule (VIEWPOINT-ADAPTATION). It takes the context type ($\mathtt{U}\langle\overline{\mathtt{d}}\rangle$), the expression that receives a method call or field access ($\mathtt{e}$), its type ($\mathtt{G}_\mathtt{e}$), and the type that should be adapted ($\mathtt{G}$), e.g., the type of a method parameter. Consider the following example. Let `J<box.boundary>` be the return type of a method `m` in a box interface `I` and let `x` be a variable typed as `I<box.local>`. The type of the method call `x.m()` is now obtained by applying the viewpoint-adaptation to the return type, resulting in type `J<x.boundary>`, because `box` is substituted with `x`. In case `I` is not a box interface, `box` is replaced by `box` again, because the owner domain of `x` is `box.local`, resulting in type `J<box.boundary>`.

4.3. Subtyping

The *subtyping relation* is given in Fig. 12; it is the transitive, reflexive closure of the `extends`-relations between interfaces and the implementing classes. Note that box interfaces and box classes are subtypes of box interfaces only. This leads to two distinct type hierarchies, and allows us to distinguish in the type system between box and non-box types. This distinction is needed to define the accessibility relation (see below). To define the subtype relation we use the function *isBoxType*($\mathtt{G}$), which returns true if $\mathtt{G}$ is a class or interface annotated with `box`.

$$\frac{\mathtt{e} = \mathtt{x} \vee \mathtt{e} = \mathtt{null}}{\mathit{validOwner}(\mathtt{e})} \qquad \frac{}{\mathit{odom}(\mathtt{U}\langle\overline{\mathtt{d}}\rangle) = \mathtt{d}_1}\ (\textsc{owner-domain}) \qquad \frac{\mathit{isBoxType}(\mathtt{L}) \vee \mathtt{L} = \bot \qquad \mathit{validOwner}(\mathtt{e})}{\mathit{boxOwner}(\mathtt{L}, \mathtt{e}) = \mathtt{e}} \qquad \frac{\neg\mathit{isBoxType}(\mathtt{C}) \qquad \mathtt{d}_1 = \mathtt{b.c}}{\mathit{boxOwner}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle, \mathtt{e}) = \mathtt{b}}$$

$$\frac{\mathtt{b} = \mathit{boxOwner}(\mathtt{G}, \mathtt{e})}{\mathit{obox}(\_, \mathtt{G}, \mathtt{e}) = \mathtt{b}}\ (\textsc{box-domain}) \qquad \frac{\mathit{isBoxType}(\mathtt{G}) \qquad \neg\mathit{validOwner}(\mathtt{e})}{\mathit{obox}(\_, \mathtt{G}, \mathtt{e}) = \ ?}\ (\textsc{box-invalidowner}) \qquad \frac{\neg\mathit{isBoxType}(\mathtt{G}) \qquad \mathit{odom}(\mathtt{G}) = \alpha}{\mathit{obox}(\alpha, \mathtt{G}, \_) = \mathtt{box}}\ (\textsc{box-same})$$

$$\frac{\mathtt{G}' = \mathtt{G}[\mathit{obox}(\mathtt{d}_1, \mathtt{G}_e, \mathtt{e})/\mathtt{box}]}{\mathtt{U}\langle\overline{\mathtt{d}}\rangle; \mathtt{G}_e; \mathtt{e} \vdash_o \mathtt{G} \triangleright \mathtt{G}'}\ (\textsc{viewpoint-adaptation})$$

Fig. 11. Functions to translate the box keyword to the corresponding user context.

$$\frac{}{\bot <: \mathtt{G}}\ (\textsc{s-null}) \qquad \frac{}{\mathtt{G} <: \mathtt{G}}\ (\textsc{s-refl}) \qquad \frac{\mathtt{G}_1 <: \mathtt{G}_2 \qquad \mathtt{G}_2 <: \mathtt{G}_3}{\mathtt{G}_1 <: \mathtt{G}_3}\ (\textsc{s-trans}) \qquad \frac{\ldots\ \mathtt{interface}\ \mathtt{I}\langle\overline{\alpha}\rangle\ \mathtt{extends}\ \overline{\mathtt{N}}\ \ldots \qquad \mathtt{N} \in \overline{\mathtt{N}}}{\mathtt{I}\langle\overline{\mathtt{d}}\rangle <: [\overline{\mathtt{d}}/\overline{\alpha}]\mathtt{N}}\ (\textsc{s-extends})$$

$$\frac{\ldots\ \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{N}\ \ldots}{\mathtt{C}\langle\overline{\mathtt{d}}\rangle <: [\overline{\mathtt{d}}/\overline{\alpha}]\mathtt{N}}\ (\textsc{s-implements})$$

Fig. 12. Subtyping rules.

$$\frac{\Gamma \vdash_o \diamond \qquad \mathtt{x} \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash_o \mathtt{N}}{\Gamma, \mathtt{x} : \mathtt{N} \vdash_o \diamond}\ (\textsc{v-env-var}) \qquad \frac{\mathit{obox}(\_, \mathtt{G}, \_) = \mathtt{box} \vee \mathit{isBoxType}(\mathtt{G})}{\mathtt{this} : \mathtt{G} \vdash_o \diamond}\ (\textsc{v-env-this})$$

Fig. 13. Well-formed environments.

4.4. Well-formed environments

The rules shown in Fig. 13 define well-formed environments. They state that variable names must be distinct and all variables are well-typed. In addition, they ensure that the context type and the context box owner are consistent. This mainly means that if the context type is not a box type then the box owner of that type must be the box owner given in the context. All type environments constructed by rules of our type system have the implicit assumption that they are well-formed according to this notion.

4.5. Accessibility relation

The first key element of the type system is the *accessibility relation* on domains presented in Fig. 14. Judgments of the form $\Gamma \vdash_o \mathtt{d}_1 \rightarrow \mathtt{d}_2$ tell us that domain $\mathtt{d}_1$ can access domain $\mathtt{d}_2$ in the given context, meaning that all objects in $\mathtt{d}_1$ can access all objects in domain $\mathtt{d}_2$. The relation formalizes the accessibility among domains depending on the ownership hierarchy of boxes. A domain has access to itself (A-REFL). The domains with the same owner, i.e., belonging to the same box, can access each other (A-OWNER). A domain can always access the global domain (A-GLOBAL) (a domain with owner `null`). The rules (A-PARAM) to (A-PARAM4) relate the domains of the current context type to the domains of the current box. Rule (A-PARAM) states that the domains of the current box can access the domains of the context type and Rule (A-PARAM2) says that the owning domain, i.e., d_1, has access to all parameter domains. These two rules essentially specify that it is impossible to pass domains through other domains without following the box hierarchy. The owning domain of the context type, i.e., the domain of the `this` object, has access to the local domain of the current box if it is not a box type (A-PARAM3). Independent of the type of the context, the owning domain can always access the boundary domain of the current box (A-PARAM4). This rule applies if the current box is an inner box of the box containing the context type and local domains of inner boxes are protected against the access from surrounding boxes. A domain can always access the boundary domain of a box, which it can access (A-BOUNDARY), and the boundary domain has always access to the owning domain of the box (A-BOUNDARY2).

$$\frac{}{\Gamma \vdash_o \mathtt{d} \to \mathtt{d}}\ (\text{A-REFL}) \qquad \frac{}{\Gamma \vdash_o \mathtt{box.c}_1 \to \mathtt{box.c}_2}\ (\text{A-OWNER}) \qquad \frac{}{\Gamma \vdash_o \mathtt{d} \to \mathtt{null.c}}\ (\text{A-GLOBAL})$$

$$\frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{this})}{\Gamma \vdash_o \mathtt{box.c} \to \overline{\mathtt{d}}}\ (\text{A-PARAM}) \qquad \frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{this})}{\Gamma \vdash_o \mathtt{d}_1 \to \overline{\mathtt{d}}}\ (\text{A-PARAM-2})$$

$$\frac{\neg \mathit{isBoxType}(\mathtt{U}) \qquad \mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{this})}{\Gamma \vdash_o \mathtt{d}_1 \to \mathtt{box.local}}\ (\text{A-PARAM-3}) \qquad \frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{this})}{\Gamma \vdash_o \mathtt{d}_1 \to \mathtt{box.boundary}}\ (\text{A-PARAM-4})$$

$$\frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{x}) \qquad \Gamma \vdash_o \mathtt{d} \to \mathtt{d}_1}{\Gamma \vdash_o \mathtt{d} \to \mathtt{x.boundary}}\ (\text{A-BOUNDARY}) \qquad \frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{x})}{\Gamma \vdash_o \mathtt{x.boundary} \to \mathtt{d}_1}\ (\text{A-BOUNDARY-2})$$

Fig. 14. Accessibility relation.

$$\frac{}{\Gamma \vdash_o \mathtt{box.c}}\ (\text{V-DOMAIN-BOX}) \qquad \frac{}{\Gamma \vdash_o \mathtt{null.c}}\ (\text{V-DOMAIN-NULL}) \qquad \frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{this})}{\Gamma \vdash_o \mathtt{d}_i}\ (\text{V-DOMAIN-PARAM})$$

$$\frac{\mathtt{U}\langle\overline{\mathtt{d}}\rangle = \Gamma(\mathtt{x}) \qquad \mathit{isBoxType}(\mathtt{U})}{\Gamma \vdash_o \mathtt{x.boundary}}\ (\text{V-DOMAIN-VAR})$$

$$\frac{\Gamma \vdash_o \overline{\mathtt{d}} \qquad \Gamma \vdash_o \mathtt{d}_1 \to \overline{\mathtt{d}} \qquad \Gamma \vdash_o \mathtt{box.c} \to \mathtt{d}_1 \qquad |\mathit{params}(\mathtt{U})| = |\overline{\mathtt{d}}|}{\Gamma \vdash_o \mathtt{U}\langle\overline{\mathtt{d}}\rangle}\ (\text{V-TYPE})$$

Fig. 15. Valid domains and types.

4.6. Valid domains and valid types

The second key element of the ownership type system is the definition of *valid domains and types* shown in Fig. 15. The notion of validity is strongly related to the accessibility relation. The most important rule is (V-TYPE), which guarantees that breaking encapsulation by passing domains, that are not accessible by the first parameter, as parameters i.e., the domain of `this` is impossible. This rule corresponds to ownership nesting rules known from other ownership type systems [58].

4.7. Typing rules for programs, interfaces and classes

The typing rules for the ownership type system are mostly standard. Fig. 16 shows the typing rules for programs, interfaces, classes and methods.

The (T-PROG) rule requires some explanation. The initial expression `e` is typed under the context type `Global` ⟨`null.local`⟩ (recall that, as explained at the beginning of Section 4, the context type information is provided through the type assumed for `this`). We just assume that `Global` is some predefined interface without any methods. In addition, all other elements of the program must be well-typed, and some sanity conditions must hold. Interfaces are well-typed if the declared method headers are well-typed. A class or box class is well-typed if all fields have valid types in the class `C`, and all methods are well-typed in the context of the class `C`. This rule instantiates the context used in the rules for expressions and to test validity of types and accessibility of domains. Methods are typed as usual. The types appearing in the method signature must be valid in the current context, and the type of the body expression must be a subtype of the declared return type.

4.8. Typing rules for expressions

The typing rules for expressions are shown in Fig. 17. The non-standard rules are (T-FIELD) and (T-INVK). Both rules exploit rule (VIEWPOINT-ADAPTATION) (see Fig. 11) to adapt the declared type to the application context.

To type `this.f`, we have to find the type for `this`, to look up the type of the field `f` (done by *field*), and then to translate the looked up type into the current context. The translated type has to be type correct as well. The rule for calls look similar, but translates the declared parameter types.

$$\text{(T-PROG)}\quad \frac{\vdash_o \overline{\texttt{ID}} \qquad \vdash_o \overline{\texttt{CD}} \qquad \texttt{this} \notin \texttt{e} \qquad \texttt{this} : \texttt{Global}\langle\texttt{null.local}\rangle \vdash_o \texttt{e} : \texttt{N}}{\vdash_o \overline{\texttt{ID}}\ \overline{\texttt{CD}}\ \texttt{e} : \texttt{N}}$$

$$\text{(T-INTERFACE)}\quad \frac{\texttt{this} : \texttt{I}\langle\overline{\alpha}\rangle \vdash_o \overline{\texttt{N}} \qquad \texttt{I}\langle\overline{\alpha}\rangle \vdash_o \overline{\texttt{S}} \qquad \forall \texttt{I}'\langle\overline{\texttt{d}}\rangle \in \overline{\texttt{N}} : (\mathit{isBoxType}(\texttt{I}) \Leftrightarrow \mathit{isBoxType}(\texttt{I}')) \wedge \alpha_1 = \texttt{d}_1 \qquad \mathit{mSig}(\texttt{I}\langle\overline{\alpha}\rangle)\ \textbf{wf}}{\vdash_o \ldots\ \texttt{interface}\ \texttt{I}\langle\overline{\alpha}\rangle\ \texttt{extends}\ \overline{\texttt{N}}\ \{\ \overline{\texttt{S}}\ \}}$$

$$\text{(T-METHOD-HEADER)}\quad \frac{\texttt{this} : \texttt{N}' \vdash_o \texttt{N} \qquad \texttt{this} : \texttt{N}' \vdash_o \overline{\texttt{N}}}{\texttt{N}' \vdash_o \texttt{N}\ \texttt{m}(\overline{\texttt{N}}\ \overline{\texttt{x}})}$$

$$\text{(T-CLASS)}\quad \frac{\mathit{mSig}(\texttt{I}\langle\overline{\texttt{d}}\rangle) \subseteq \mathit{mSig}(\texttt{C}\langle\overline{\alpha}\rangle) \qquad \texttt{this} : \texttt{C}\langle\overline{\alpha}\rangle \vdash_o \texttt{I}\langle\overline{\texttt{d}}\rangle \qquad \texttt{C}\langle\overline{\alpha}\rangle \vdash_o \overline{\texttt{M}} \qquad \alpha_1 = \texttt{d}_1 \qquad \texttt{this} : \texttt{C}\langle\overline{\alpha}\rangle \vdash_o \overline{\texttt{N}} \qquad \mathit{isBoxType}(\texttt{C}) \Leftrightarrow \mathit{isBoxType}(\texttt{I})}{\vdash_o \ldots\ \texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{I}\langle\overline{\texttt{d}}\rangle\ \texttt{by}\ \{\overline{\texttt{N}}\ \overline{\texttt{f}};\ \bullet;\ \overline{\texttt{M}}\}\ \ \{\overline{\texttt{N}}\ \overline{\texttt{f}};\ \}}$$

$$\text{(T-METHOD)}\quad \frac{\texttt{S} = \texttt{N}\ \texttt{m}\ (\overline{\texttt{N}}\ \overline{\texttt{x}}) \qquad \texttt{this} : \texttt{G} \vdash_o \texttt{N} \cup \overline{\texttt{N}} \qquad \texttt{this} : \texttt{G}, \overline{\texttt{x}} : \overline{\texttt{N}} \vdash_o \texttt{e} : \texttt{L} \qquad \texttt{L} <: \texttt{N}}{\texttt{G} \vdash_o \texttt{S}\{\texttt{return e}\}}$$

Fig. 16. Program, interface, class, and method typing.

$$\text{(T-NULL)}\quad \frac{}{\Gamma \vdash_o \texttt{null} : \bot} \qquad \text{(T-VAR)}\quad \frac{\texttt{x} : \texttt{G} \in \Gamma}{\Gamma \vdash_o \texttt{x} : \texttt{G}}$$

$$\text{(T-FIELD)}\quad \frac{\Gamma \vdash_o \texttt{this} : \texttt{G} \qquad \texttt{N}_\texttt{f} = \mathit{field}(\texttt{G}, \texttt{f}) \qquad \texttt{G}; \texttt{G}; \texttt{this} \vdash_o \texttt{N}_\texttt{f} \triangleright \texttt{N}'_\texttt{f} \qquad \Gamma \vdash_o \texttt{N}'_\texttt{f}}{\Gamma \vdash_o \texttt{this.f} : \texttt{N}'_\texttt{f}}$$

$$\text{(T-FIELD-UP)}\quad \frac{\Gamma \vdash_o \texttt{this.f} : \texttt{G} \qquad \Gamma \vdash_o \texttt{e} : \texttt{L} \qquad \texttt{L} <: \texttt{G}}{\Gamma \vdash_o \texttt{this.f} = \texttt{e} : \texttt{G}}$$

$$\text{(T-INVK)}\quad \frac{\begin{array}{c}\Gamma \vdash_o \texttt{e} : \texttt{G}_\texttt{e} \qquad \Gamma \vdash_o \overline{\texttt{e}} : \overline{\texttt{L}} \qquad \mathit{mSig}(\texttt{G}_\texttt{e}) = \ \ldots\ \texttt{G}_\texttt{m}\ \texttt{m}(\overline{\texttt{G}}_\texttt{m})\ \ldots \\ \Gamma(\texttt{this}); \texttt{G}_\texttt{e}; \texttt{e} \vdash_o \texttt{G}_\texttt{m} \triangleright \texttt{G}'_\texttt{m} \qquad \Gamma(\texttt{this}); \texttt{G}_\texttt{e}; \texttt{e} \vdash_o \overline{\texttt{G}}_\texttt{m} \triangleright \overline{\texttt{G}}'_\texttt{m} \qquad \Gamma \vdash_o \texttt{G}'_\texttt{m} \qquad \Gamma \vdash_o \overline{\texttt{G}}'_\texttt{m} \qquad \overline{\texttt{L}} <: \overline{\texttt{G}}'_\texttt{m}\end{array}}{\Gamma \vdash_o \texttt{e.m}(\overline{\texttt{e}}) : \texttt{G}'_\texttt{m}}$$

$$\text{(T-NEW-CLASS)}\quad \frac{\ldots\ \texttt{class}\ \texttt{C}\langle\overline{\alpha}\rangle\ \texttt{implements}\ \texttt{N}\ \ldots \qquad \Gamma \vdash_o \texttt{C}\langle\overline{\texttt{d}}\rangle}{\Gamma \vdash_o \texttt{new}\ \texttt{C}\langle\overline{\texttt{d}}\rangle : \texttt{N}[\overline{\texttt{d}}/\overline{\alpha}]} \qquad \text{(T-CAST)}\quad \frac{\Gamma \vdash_o \texttt{N} \qquad \Gamma \vdash_o \texttt{e} : \texttt{L} \qquad \texttt{N} <: \texttt{L} \vee \texttt{L} <: \texttt{N}}{\Gamma \vdash_o (\texttt{N})\texttt{e} : \texttt{N}}$$

Fig. 17. Expression type rules.

5. Constraint-based ownership typing for IFBTJ

In this section, we present the constraint-based ownership type system for IFBTJ. The constraint-based type system analyzes each trait definition in isolation from the classes or traits that use it (see Section 1). The idea is to type-check (the methods provided by) a trait definition by using ownership type-constraints to collect the ownership type-checks that require to know the class C of the `this` object (that is, the class C composed by using the trait). These constraints will then be checked when type-checking the classes that are composed by using the trait.

As we will formally state in Section 6: (i) for FIFBTJ programs, the constraint-based ownership type system is equivalent to the ownership type system given in Section 4 (that is, if an FIFBTJ program type-checks with respect to one of the two systems, then it will type-check also with respect to the other); and (ii) the constraint-based ownership type system conforms to the flattening principle (that is, if an IFBTJ program type-checks then also its flattened version will type-check).

5.1. Overview

The constraint-based type system collects, for each method definition in the trait, the constraints on the use of `this` within the method body. In particular, each method

$$\mathtt{M} = \mathtt{N}_0\ \mathtt{m}\ (\overline{\mathtt{N}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e;}\ \} \quad \in \quad \overline{\mathtt{M}}$$

defined within a basic trait expression $\{\,\overline{\mathtt{F}};\ \overline{\mathtt{S}};\ \overline{\mathtt{M}}\,\}$ is type-checked by assuming for `this` the structural type $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle$, where $\overline{\mathtt{F}}$ and $\overline{\sigma} = \mathit{toSig}(\overline{\mathtt{S}}) \cdot \mathit{toSig}(\overline{\mathtt{M}})$ are the required fields and the signatures of the required/provided methods of the basic trait expression, respectively. The typing judgment for method definitions has the following form:

$$\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash_{\mathtt{co}} \mathtt{M} : \mu$$

which is to be read "assuming the structural type $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle$ for `this`, the method `M` has constraint-based type μ", where

$$\mu = \mathtt{N}_0\ \mathtt{m}(\overline{\mathtt{N}})\ \mid\ \langle\ \overline{\mathtt{F}}'\ \mid\ \overline{\sigma}'\ \rangle\ \mid\ (\overline{\mathtt{x}}, \Phi)$$

is such that

1. $\mathtt{N}_0\ \mathtt{m}\ (\overline{\mathtt{N}})$ is the signature of the method;
2. the pair $\langle\ \overline{\mathtt{F}}'\ \mid\ \overline{\sigma}'\ \rangle$ specifies that the body of the method (the expression `e`) selects the fields $\overline{\mathtt{F}}'$ ($\subseteq \overline{\mathtt{F}}$) and the methods with signatures $\overline{\sigma}'$ ($\subseteq \overline{\sigma}$) on `this`;
3. $\overline{\mathtt{x}}$ are the names of the formal parameter of the method; and
4. Φ is a the set of constraints representing the checks that must be performed when type-checking the classes that use a trait providing the method.

In particular, the constraints in Φ represent the view-point adaptations, the valid ownership annotation checks, the subtype checks, and the method signature lookups that are performed by the ownership type system for FIFBTJ (see Section 4) when type-checking the method definition `M` within a class definition

$$\ldots\ \mathtt{class\ C}\langle\alpha_1\alpha_2\cdots\alpha_n\rangle\ \mathtt{implements\ I}\langle\alpha_1\mathtt{d}_2\cdots\mathtt{d}_n\rangle\ \mathtt{by}\ \{\,\overline{\mathtt{F}};\ \bullet;\ \overline{\mathtt{M}}\,\}\ \{\,\overline{\mathtt{N}}\ \overline{\mathtt{f}};\ \}$$

The constraints, collected by the ownership type system when analyzing (the method definitions occurring in) a trait definition, will be checked when analyzing the definition of the classes that use that trait.

Ownership types are responsible for most of the constraints. In particular, when ownership types are not considered (like, e.g., in [13]), the pair $(\overline{\mathtt{x}}, \Phi)$ (described by items 3 and 4 above) can be replaced by the sequence $\overline{\mathtt{I}}$ of the interfaces that, according to the use of this in the body of the method, must be implemented by the class of the this object.

The following example illustrates a situation where it is not possible to perform the ownership type checks when analyzing a trait definition.

```
interface H⟨α⟩ extends • { J⟨box.boundary⟩ m(); }

interface J⟨α⟩ extends • {···}

interface I⟨α, γ⟩ extends • { J⟨γ⟩ foo(H⟨box.local⟩ x); }

trait T⟨α, β⟩ is {
  J⟨β⟩ f;
  J⟨β⟩ foo(H⟨box.local⟩ x) {return this.f = x.m(); }   // not possible to perform ownership type check here
}

class C1⟨α⟩ implements I⟨α, box.boundary⟩ by T⟨α, box.boundary⟩ { J⟨box.boundary⟩ f; }

class C2⟨α⟩ implements I⟨α, α⟩ by T⟨α, α⟩ { J⟨α⟩ f; }
```

The interface `H` has a single method `m` that has return type `J<box.boundary>`, where `J` is another interface. The interface `I` has a method `foo` that has a parameter `J<box.local> x`. The trait $\mathtt{T}\langle\alpha, \beta\rangle$ requires a field $\mathtt{J}\langle\beta\rangle$ `f` and implements method `foo` by assigning the result of the method call `x.m()` to `f` and returns that value.

Since the return type of `x.m()` is $\mathtt{J}\langle\mathtt{box.boundary}\rangle$ (after viewpoint adaptation), the type correctness of the assignment depends on the instantiation of the domain parameter β of trait `T`.

The classes $\mathtt{C}_1$ and $\mathtt{C}_2$ use the trait `T` with two different domain parameter instantiations. Class $\mathtt{C}_1$ uses `box.boundary` for β and class $\mathtt{C}_2$ uses α.

The flattened versions of classes $\mathtt{C}_1$ and $\mathtt{C}_2$ are as follows.

```
class C1⟨α⟩ implements I⟨α, box.boundary⟩ {
  J⟨box.boundary⟩ f;
  J⟨box.boundary⟩ foo(H⟨box.local⟩ x) {return this.f = x.m(); }   // ownership type check succeeds
}

class C2⟨α⟩ implements I⟨α, α⟩ {
  J⟨α⟩ f;
  J⟨α⟩ foo(H⟨box.local⟩ x) {return this.f = x.m(); }   // ownership type check fails
}
```

Open nominal types and open types

$$\begin{array}{rcll} \mathtt{O} & ::= & \mathtt{G} \mid \mathtt{X} \mid \chi & \text{open nominal types} \\ \mathtt{R} & ::= & \mathtt{O} \mid \bot & \text{open types} \end{array}$$

Constraints

$$\begin{array}{rcll} \psi & ::= & \textbf{isValid}(\mathtt{O}) & \text{type } \mathtt{O} \text{ has valid domain annotations} \\ & \mid & \textbf{vpa}(\mathtt{O},\ \mathtt{e},\ \mathtt{O}',\ \mathtt{X}) & \text{type } \mathtt{X} \text{ is the view-point adaptation of type } \mathtt{O}' \text{ and has valid domain annotations} \\ & \mid & \textbf{sub}(\mathtt{R},\ \mathtt{O}) & \mathtt{R} \text{ is a subtype of } \mathtt{O} \\ & \mid & \textbf{cast}(\mathtt{N},\ \mathtt{R}) & \text{type } \mathtt{R} \text{ can be casted to type } \mathtt{N} \\ & \mid & \textbf{mTyp}(\mathtt{O},\ \mathtt{m},\ \mathtt{X}\overline{\mathtt{X}}) & \text{source type } \mathtt{O} \text{ has method } \mathtt{m} \text{ with signature } \mathtt{X}\ \mathtt{m}(\overline{\mathtt{X}}) \end{array}$$

Rules for checking constraints satisfaction w.r.t. a type environment

$$\text{(CC-isValid)}\ \frac{\Gamma \vdash_{\mathtt{o}} \mathtt{G} \qquad \Gamma \vdash_{\mathtt{co}} \Phi}{\Gamma \vdash_{\mathtt{co}} \Phi \uplus \{\textbf{isValid}(\mathtt{G})\}} \qquad \text{(CC-vpa)}\ \frac{\Gamma(\mathtt{this}); \mathtt{G_e}; \mathtt{e} \vdash_{\mathtt{o}} \mathtt{G} \triangleright \mathtt{G}' \qquad \Gamma \vdash_{\mathtt{o}} \mathtt{G}' \qquad \Gamma \vdash_{\mathtt{co}} \Phi[^{\mathtt{G}'}/_{\mathtt{X}}]}{\Gamma \vdash_{\mathtt{co}} \Phi \uplus \{\textbf{vpa}(\mathtt{G_e},\ \mathtt{e},\ \mathtt{G},\ \mathtt{X})\}}$$

$$\text{(CC-sub)}\ \frac{\mathtt{L} <: \mathtt{N} \qquad \Gamma \vdash_{\mathtt{co}} \Phi}{\Gamma \vdash_{\mathtt{co}} \Phi \uplus \{\textbf{sub}(\mathtt{L},\ \mathtt{N})\}} \qquad \text{(CC-cast)}\ \frac{\Gamma \vdash_{\mathtt{o}} \mathtt{N} \qquad \mathtt{N} <: \mathtt{L} \vee \mathtt{L} <: \mathtt{N} \qquad \Gamma \vdash_{\mathtt{co}} \Phi}{\Gamma \vdash_{\mathtt{co}} \Phi \uplus \{\textbf{cast}(\mathtt{N},\ \mathtt{L})\}}$$

$$\text{(CC-mTyp)}\ \frac{\mathit{mSig}(\mathtt{N}, \mathtt{m}) = \mathtt{G}\ \mathtt{m}(\overline{\mathtt{G}}) \qquad \Gamma \vdash_{\mathtt{co}} \Phi[^{\mathtt{G}\overline{\mathtt{G}}}/_{\overline{\mathtt{X}}}]}{\Gamma \vdash_{\mathtt{co}} \Phi \uplus \{\textbf{mTyp}(\mathtt{N},\ \mathtt{m},\ \overline{\mathtt{X}})\}} \qquad \text{(CC-Empty)}\ \frac{}{\Gamma \vdash_{\mathtt{co}} \emptyset}$$

Fig. 18. Open types syntax (top), constraints syntax (middle), and constraint satisfaction checking rules (bottom)

The traits have been removed and the domain parameters of the traits have been replaced by their instantiations. According to the ownership type system for FIFBTJ (see Section 4), class $\mathtt{C}_1$ is correctly typed, while class $\mathtt{C}_2$ is type incorrect. The type of `x.m()` is in both classes $\mathtt{J}\langle\mathtt{box.boundary}\rangle$, which is assigned to a field of type $\mathtt{J}\langle\mathtt{box.boundary}\rangle$ in class $\mathtt{C}_1$, but to a field with type $\mathtt{J}\langle\alpha\rangle$ in $\mathtt{C}_2$.

5.2. Constraints and constraint checking rules

The constraints, illustrated in Fig. 18 (middle), involve types and type variables, including the distinguished type variable χ. Type variables, ranged over by X, will be instantiated to nominal types when checking the constraints. The variable χ will be instantiated to the type of `this`. The metavariable O denotes either a nominal type or a type variable, while the metavariable R denotes either a type or a type variable, as illustrated in Fig. 18 (top).

The checking judgement for constraints is $\Gamma \vdash_{\mathtt{co}} \Phi$, to be read "the constraints in the set Φ are satisfied with respect to the type environment Γ". The associated typing rules are given in Fig. 18 (bottom); the operator $\uplus$ denotes the disjoint rules of set of constraints. The rules (which rely on judgments introduced in Section 4) are almost self-explanatory, according to the informal meaning given in the middle of Fig. 18. In particular,

- (CC-isValid) relies on rule (v-type) of Fig. 15;
- (CC-vpa) relies on rule (viewpoint-adaptation) of Fig. 11 and on rule (v-type) of Fig. 15;
- (CC-sub) relies on the subtyping rules, given in Fig. 12;
- (CC-cast) relies on rule (v-type) and the subtyping rules (according to the typing rule (t-cast) in Fig. 17); and
- (CC-mTyp) relies on the signature lookup function $\mathit{mSig}(\cdot, \cdot)$, given in Fig. 10.

We say that a constraint is *ground* to mean that it contains no type variables. The checking of a constraint of the form **isValid**$(\cdot)$, **sub**$(\cdot,\ \cdot)$ or **cast**$(\cdot,\ \cdot)$ can be performed only when the constraint is ground. The checking of a constraint of the form **vpa**$(\cdot,\ \cdot,\ \cdot,\ \cdot)$ or **mTyp**$(\cdot,\ \cdot,\ \cdot)$ can be performed only when the last argument contains only type variables and there are no occurrences of type variables in the other arguments; the checking causes the instantiation of all the type variables occurring in the third argument.

5.3. Auxiliary functions annVars and progVars

The constraint-based typing uses the auxiliary functions *annVars* and *progVars*, which are defined as follows:

> *annVars*(TE) returns the sequence of the annotation variables α that occur within the trait expression TE, and *progVars*($\overline{\mathtt{d}}$) returns the sequence of the variables x (possibly including `this`) that occur within the sequence of domain annotations $\overline{\mathtt{d}}$.

(CT-Program)
$$\frac{\vdash_{\mathrm{o}} \overline{\mathtt{ID}} \quad \vdash_{\mathrm{co}} \overline{\mathtt{TD}} : \cdots \quad \vdash_{\mathrm{co}} \overline{\mathtt{CD}} \quad \mathtt{this} \notin \mathtt{e} \quad \mathtt{this} : \mathtt{Global}\langle \mathtt{null.local} \rangle \vdash_{\mathrm{o}} \mathtt{e} : \mathtt{N}}{\vdash_{\mathrm{co}} \overline{\mathtt{ID}}\ \overline{\mathtt{TD}}\ \overline{\mathtt{CD}}\ \mathtt{e} : \mathtt{N}}$$

(CT-Trait)
$$\frac{\overline{\alpha} = \alpha_1, \ldots \quad \alpha_1 \vdash_{\mathrm{co}} \mathtt{TE} : \overline{\mu} \quad \mathit{annVars}(\mathtt{TE}) \subseteq \overline{\alpha}}{\vdash_{\mathrm{co}} \mathtt{trait}\ \mathtt{T}\langle \overline{\alpha} \rangle\ \mathtt{is}\ \mathtt{TE} : \overline{\mu}}$$

(CT-Class)
$$\frac{\begin{array}{c}\overline{\alpha} = \alpha_1, \alpha_2, \ldots \alpha_n \quad \overline{\mathtt{d}} = \alpha_1, \mathtt{d}_2, \ldots \mathtt{d}_n \quad \alpha_1 \vdash_{\mathrm{co}} \mathtt{TE} : \mu_1 \ldots \mu_p \quad p \geq 0 \quad \mathit{annVars}(\mathtt{TE}) \subseteq \overline{\alpha} \\ \forall i \in 1..p, \quad \mu_i = \zeta_i \triangleright \langle\, \overline{\mathtt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\mathtt{x}}^{(i)}, \Phi_i) \quad \zeta_i = \mathtt{N}_i\ (\overline{\mathtt{N}}^{(i)}) \quad \mathtt{this} : \mathtt{C}\langle \overline{\alpha} \rangle,\ \overline{\mathtt{x}}^{(i)} : \overline{\mathtt{N}}^{(i)} \vdash_{\mathrm{co}} \Phi_i[{}^{\mathtt{C}\langle \overline{\alpha} \rangle}/_{\chi}] \\ \cup_{i \in 1..p} \overline{\sigma}^{(i)} \subseteq \zeta_1 \ldots \zeta_p \quad (\cup_{i \in 1..p} \overline{\mathtt{F}}^{(i)}) = \overline{\mathtt{N}}\ \overline{\mathtt{f}} \\ \mathit{mSig}(\mathtt{I}\langle \overline{\mathtt{d}} \rangle) \subseteq \zeta_1 \ldots \zeta_p \quad \mathtt{this} : \mathtt{C}\langle \overline{\alpha} \rangle \vdash_{\mathrm{o}} \mathtt{I}\langle \overline{\mathtt{d}} \rangle \quad \mathtt{this} : \mathtt{C}\langle \overline{\alpha} \rangle \vdash_{\mathrm{o}} \overline{\mathtt{N}} \quad \mathit{isBoxType}(\mathtt{C}) \Leftrightarrow \mathit{isBoxType}(\mathtt{I})\end{array}}{\vdash_{\mathrm{co}} \ldots \mathtt{class}\ \mathtt{C}\langle \overline{\alpha} \rangle\ \mathtt{implements}\ \mathtt{I}\langle \overline{\mathtt{d}} \rangle\ \mathtt{by}\ \mathtt{TE}\ \{\, \overline{\mathtt{N}}\ \overline{\mathtt{f}};\ \}}$$

(CT-TEBasic)
$$\frac{\begin{array}{c}\mathit{mSig}(\overline{\mathtt{S}}) = \overline{\sigma} \\ \mathit{mSig}(\mathtt{M}_1 \ldots \mathtt{M}_p) = \zeta_1 \ldots \zeta_p \quad p \geq 0 \quad \forall i \in 1..p, \quad \langle\, \overline{\mathtt{F}} \triangleright \overline{\sigma} \cdot \zeta_1 \ldots \zeta_p \,\rangle \vdash_{\mathrm{co}} \mathtt{M}_i : \mu_i \quad \mu_i = \zeta_i \triangleright \langle\, \overline{\mathtt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\mathtt{x}}^{(i)}, \Phi_i) \\ \overline{\mathtt{F}} = \cup_{i \in 1..p} \overline{\mathtt{F}}^{(i)} \quad \overline{\sigma} = \mathit{exclude}((\cup_{i \in 1..p} \overline{\sigma}^{(i)}), \mathit{names}(\zeta_1 \ldots \zeta_p))\end{array}}{\alpha \vdash_{\mathrm{co}} \{\, \overline{\mathtt{F}};\ \overline{\mathtt{S}};\ \mathtt{M}_1 \ldots \mathtt{M}_p \,\} : \mu_1 \ldots \mu_p}$$

(CT-Method)
$$\frac{\begin{array}{c}\mathtt{S} = \mathtt{N}_0\ \mathtt{m}\ (\overline{\mathtt{N}}\ \overline{\mathtt{x}}) \quad \mathtt{this} : \langle\, \overline{\mathtt{F}} \triangleright \overline{\sigma} \,\rangle,\ \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash_{\mathrm{co}} \mathtt{e} : \mathtt{R} \triangleright \langle\, \overline{\mathtt{F}}' \triangleright \overline{\sigma}' \,\rangle \triangleright \Phi_0 \\ \overline{\mathtt{N}} = \mathtt{N}_1 \ldots \mathtt{N}_n \quad (n \geq 0) \quad \Phi = \Phi_0 \cup \{\mathbf{sub}(\mathtt{R},\ \mathtt{N}_0)\} \cup (\cup_{i=0..n} \{\mathbf{isValid}(\mathtt{N}_i)\})\end{array}}{\langle\, \overline{\mathtt{F}} \triangleright \overline{\sigma} \,\rangle \vdash_{\mathrm{co}} \mathtt{S}\,\{\mathtt{return}\ \mathtt{e};\ \} : \mathtt{N}_0\ \mathtt{m}(\overline{\mathtt{N}}) \triangleright \langle\, \overline{\mathtt{F}}' \triangleright \overline{\sigma}' \,\rangle \triangleright (\overline{\mathtt{x}}, \Phi)}$$

Fig. 19. Program, trait, class, basic trait expression, and method constraint-based typing rules.

5.4. Constraint-based typing rules for programs, traits, classes, basic trait expressions, and methods

Fig. 19 shows the constraint-based typing rules for programs, traits, classes, basic trait expressions, and methods. In order to understand how these rules work, it is useful to compare them with the typing rules of system $\vdash_{\mathrm{o}}$ in Fig. 16.

Rule (CT-Program) exploits the typing rules of system $\vdash_{\mathrm{o}}$ to type the interface definitions and the main expression of the program. The typing rule for trait definitions, (CT-Trait), assigns to a trait definition $\mathtt{trait}\ \mathtt{T}\langle\overline{\alpha}\rangle\ \mathtt{is}\ \mathtt{TE}$ the sequence $\overline{\mu}$ of the constraint-based types of the methods provided by the trait TE (the structure of the constraint-based type μ of a method has been illustrated at the beginning of Section 5). Note that, since the trait reuse graph is acyclic, no uses of T may be encountered when typing TE. The typing rule for class definitions, (CT-Class), exploits the constraint-based types inferred for the methods provided by the trait expression TE and the checking rules for constraints (illustrated in Section 5.2) to perform the same checks that would be performed by rule (T-class) of Fig. 16 when type-checking a class that contains the methods provided by TE. The typing rule for basic trait expressions, (CT-TEBasic), infers a constraint-based type for each provided method by assuming for this the structural type $\langle\, \overline{\mathtt{F}} \triangleright \overline{\sigma} \,\rangle$, where $\overline{\mathtt{F}}$ and $\overline{\sigma} = \mathit{toSig}(\overline{\mathtt{S}}) \cdot \mathit{toSig}(\overline{\mathtt{M}})$ are the required fields and the signatures of the required/provided methods of the basic trait expression, respectively. This rule also checks that each of the declared as required fields/methods is used in at least one of the provided methods. The typing rule for methods, (CT-Method), infers a constraint-based type for the body of the method and then builds the constraint-based type for the method by adding to the constraints inferred for the body of the methods the constraints expressing that the type of the body of the method must be a subtype of the declared return type and that the return and parameter types have valid domain annotations.

5.5. Constraint-based typing rules for non-basic trait expressions

Fig. 20 shows the constraint-based typing rules for non-basic trait expressions. The rule for trait expression $\mathtt{T}\langle\alpha\overline{\mathtt{d}}\rangle$, (CT-TEName), looks up the typing of the trait definition $\mathtt{trait}\ \mathtt{T}\langle\overline{\beta}\rangle$ ... and specializes the sequence of constraint-based types of the provided methods by instantiating the annotation formal parameters $\overline{\beta}$ to the actual parameters $\alpha\overline{\mathtt{d}}$. The rule for symmetric sum, (CT-TESum), assigns to the composed trait the concatenation of the sequences of constraint-based method types inferred for the summed traits; thus, it checks that there are no conflicts among the methods provided by the summed traits (since $\overline{\mu}_1 \ldots \overline{\mu}_{p+q}$ is a sequence of named elements, it does not contain duplicated names). It also checks that there are

$$\text{(CT-TEname)}\quad \frac{\vdash_{\text{co}} \texttt{trait}\ \texttt{T}\langle\overline{\beta}\rangle\ \ldots : \overline{\mu} \qquad \mathit{progVars}(\overline{\texttt{d}}) = \bullet}{\alpha \vdash_{\text{co}} \texttt{T}\langle\alpha\overline{\texttt{d}}\rangle : \overline{\mu}[{}^{\alpha\overline{\texttt{d}}}/_{\overline{\beta}}]}$$

$$\text{(CT-TEsum)}\quad \frac{\begin{array}{c}\alpha \vdash_{\text{co}} \texttt{TE}_1 : \mu_1 \ldots \mu_p \qquad \alpha \vdash_{\text{co}} \texttt{TE}_2 : \mu_{p+1} \ldots \mu_{p+q} \\ \forall i \in 1..p+q, \qquad \mu_i = \zeta_i \triangleright \langle\, \overline{\texttt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\texttt{x}}^{(i)}, \Phi_i) \qquad \cup_{i\in 1..p+q}\overline{\texttt{F}}^{(i)}\ \textbf{ok} \qquad \zeta_1 \ldots \zeta_{p+q} \cup (\cup_{i\in 1..p+q}\overline{\sigma}^{(i)})\ \textbf{ok}\end{array}}{\alpha \vdash_{\text{co}} \texttt{TE}_1 + \texttt{TE}_2 : \mu_1 \ldots \mu_{p+q}}$$

$$\text{(CT-TEexclude)}\quad \frac{\alpha \vdash_{\text{co}} \texttt{TE} : \overline{\mu} \qquad \texttt{m} \in \mathit{names}(\overline{\mu})}{\alpha \vdash_{\text{co}} \texttt{TE}\,[\texttt{exclude}\ \texttt{m}] : \mathit{exclude}(\overline{\mu}, \texttt{m})}$$

$$\text{(CT-TEalias)}\quad \frac{\begin{array}{c}\alpha \vdash_{\text{co}} \texttt{TE} : \mu_1 \ldots\ldots \mu_n \\ 1 \le p \le n \qquad \mathit{names}(\mu_p) = \texttt{m} \qquad \texttt{m}' \notin \mathit{names}(\mu_1 \ldots \mu_n) \qquad \forall i \in 1..n, \qquad \mu_i = \zeta_i \triangleright \langle\, \overline{\texttt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\texttt{x}}^{(i)}, \Phi_i) \\ \mu = \zeta_p[{}^{\texttt{m}'}/_{\texttt{m}}] \triangleright \langle\, \overline{\texttt{F}}^{(p)} \triangleright \overline{\sigma}^{(p)} \,\rangle \triangleright (\overline{\texttt{x}}^{(p)}, \Phi_p) \qquad \zeta_p[{}^{\texttt{m}'}/_{\texttt{m}}] \cup (\cup_{i\in 1..n}\overline{\sigma}^{(i)})\ \textbf{ok}\end{array}}{\alpha \vdash_{\text{co}} \texttt{TE}\,[\texttt{m}\ \texttt{aliasAs}\ \texttt{m}'] : \mu_1 \ldots \mu_n \cdot \mu}$$

$$\text{(CT-TErenameM)}\quad \frac{\begin{array}{c}(\texttt{m} \in \mathit{names}(\mu_1 \ldots\ldots \mu_n) \quad \text{implies} \quad \texttt{m}' \notin \mathit{names}(\mu_1 \ldots\ldots \mu_n)) \\ \alpha \vdash_{\text{co}} \texttt{TE} : \mu_1 \ldots\ldots \mu_n \qquad \forall i \in 1..n, \qquad \mu_i = \zeta_i \triangleright \langle\, \overline{\texttt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\texttt{x}}^{(i)}, \Phi_i) \qquad \overline{\sigma} = \zeta_1 \ldots \zeta_n \cup (\cup_{i\in 1..n}\overline{\sigma}^{(i)}) \\ \texttt{m} \in \mathit{names}(\overline{\sigma}) \qquad \overline{\sigma}[{}^{\texttt{m}'}/_{\texttt{m}}]\ \textbf{ok} \qquad \forall i \in 1..n, \qquad \mu'_i = \zeta_i[{}^{\texttt{m}'}/_{\texttt{m}}] \triangleright \langle\, \overline{\texttt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)}[{}^{\texttt{m}'}/_{\texttt{m}}] \,\rangle \triangleright (\overline{\texttt{x}}^{(i)}, \Phi_i)\end{array}}{\alpha \vdash_{\text{co}} \texttt{TE}\,[\texttt{rename}\ \texttt{m}\ \texttt{to}\ \texttt{m}'] : \mu'_1 \ldots \mu'_n}$$

$$\text{(CT-TErenameF)}\quad \frac{\begin{array}{c}\alpha \vdash_{\text{co}} \texttt{TE} : \mu_1 \ldots\ldots \mu_n \qquad \forall i \in 1..n, \qquad \mu_i = \zeta_i \triangleright \langle\, \overline{\texttt{F}}^{(i)} \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\texttt{x}}^{(i)}, \Phi_i) \\ \overline{\texttt{F}} = \overline{\texttt{F}}^{(1)} \cup \cdots \cup \overline{\texttt{F}}^{(n)} \qquad \texttt{f} \in \mathit{names}(\overline{\texttt{F}}) \qquad \overline{\texttt{F}}[{}^{\texttt{f}'}/_{\texttt{f}}]\ \textbf{ok} \qquad \forall i \in 1..n, \qquad \mu'_i = \zeta_i \triangleright \langle\, \overline{\texttt{F}}^{(i)}[{}^{\texttt{f}'}/_{\texttt{f}}] \triangleright \overline{\sigma}^{(i)} \,\rangle \triangleright (\overline{\texttt{x}}^{(i)}, \Phi_i)\end{array}}{\alpha \vdash_{\text{co}} \texttt{TE}\,[\texttt{rename}\ \texttt{f}\ \texttt{to}\ \texttt{f}'] : \mu'_1 \ldots \mu'_n}$$

Fig. 20. Non-basic trait expression constraint-based typing rules.

no conflicts among the fields required by the summed traits ($\cup_{i\in 1..p+q}\overline{\texttt{F}}^{(i)}$ holds) and among the provided methods ($\zeta_1 \ldots \zeta_{p+q}$) and the required methods ($\cup_{i\in 1..p+q}\overline{\sigma}^{(i)}$). The rule for method exclusion, (CT-TEexclude), removes the constraint-based type of the excluded method. The rule for method aliasing, (CT-TEalias), checks that the method to be aliased exists, that the name of the alias does not create conflicts, and adds the constraint-based type of the alias method. The typing rule for method renaming, (CT-TErenameM), checks that the method to be renamed exists, that the new name does not create conflicts, and replaces all the occurrences of the name of the method to be renamed with the new name. The typing rule for required field renaming, (CT-TErenameF), is similar.

5.6. Constraint-based typing rules for expressions

Fig. 21 shows the constraint-based typing rules for expressions. In order to understand how these rules work, it is useful to compare them with the typing rules for expressions of system $\vdash_{\text{o}}$ in Fig. 17.

The rule for `null`, (CT-Null), is straightforward; no constraints have to be collected. The rule for variables, (CT-Var), uses the distinguished type variable χ when $\texttt{x} = \texttt{this}$, and looks up in the environment Δ when $\texttt{x} \neq \texttt{this}$; no constraints have to be collected. The rule for field selection, (CT-Field), looks up the structural type of `this` in Δ, extracts the type `N` of `f`, and collects the constraint that `this` must have a field `f` of type `N`. Note that, since the expression `e` to be checked occurs in the body of a method `M`, the structural type $\Delta(\texttt{this})$ contains the required fields declaration of a basic trait expression $\{\,\overline{\texttt{F}};\ \overline{\texttt{S}};\ \overline{\texttt{M}}\,\}$ such that $\texttt{M} \in \overline{\texttt{M}}$. The constraints collected by means of rule (CT-Field) are a subset on the assumptions $\overline{\texttt{F}}$ provided by $\Gamma(\texttt{this})$: they describe the fields that are selected on `this` by the checked expression. A constraint expressing that the view-point adaptation of the type of the field must have valid annotations is collected. The rule for field assignment, (CT-FieldUp), builds the constraint-based type for the assignment $\texttt{this.f} = \texttt{e}$ by adding to the constraints inferred for `this.f` and `e` the constraint expressing that the type of `e` must be a subtype of the type of `this.f`. In the rule for method invocation on `this`, (Ct-ThisInvk), the actual parameters ($\texttt{e}_1, \ldots, \texttt{e}_n$) are checked and the inferred constraints are collected in the conclusion of the rule. Then, the signature ζ of `m` is extracted from the sequence of signatures $\overline{\sigma}$ in the type assumed

$$\text{(CT-Null)}\quad \frac{}{\Delta \vdash_{\text{co}} \texttt{null} : \bot \mid \langle \bullet \mid \bullet \rangle \mid \{\}}$$

$$\text{(CT-Var)}\quad \frac{\texttt{O} = \begin{cases} \chi & \text{if } \texttt{x} = \texttt{this} \\ \Delta(\texttt{x}) & \text{if } \texttt{x} \neq \texttt{this} \end{cases}}{\Delta \vdash_{\text{co}} \texttt{x} : \texttt{O} \mid \langle \bullet \mid \bullet \rangle \mid \{\}}$$

$$\text{(CT-Field)}\quad \frac{\Delta \vdash_{\text{co}} \texttt{this} : \chi \mid \langle \bullet \mid \bullet \rangle \mid \{\} \qquad \Delta(\texttt{this}) = \langle \overline{\texttt{F}} \mid \cdots \rangle \qquad \mathit{choose}(\overline{\texttt{F}}, \texttt{f}) = \texttt{N f} \qquad \texttt{X fresh}}{\Delta \vdash_{\text{co}} \texttt{this.f} : \texttt{X} \mid \langle \texttt{N f} \mid \bullet \rangle \mid \{\mathbf{vpa}(\chi, \texttt{this}, \texttt{N}, \texttt{X})\}}$$

$$\text{(CT-FieldUp)}\quad \frac{\Delta \vdash_{\text{co}} \texttt{this.f} : \texttt{X} \mid \langle \texttt{F} \mid \bullet \rangle \mid \Phi \qquad \Delta \vdash_{\text{co}} \texttt{e} : \texttt{R} \mid \langle \overline{\texttt{F}} \mid \overline{\sigma} \rangle \mid \Phi}{\Delta \vdash_{\text{co}} \texttt{this.f} = \texttt{e} : \texttt{X} \mid \langle \texttt{F} \cup \overline{\texttt{F}} \mid \overline{\sigma} \rangle \mid \Phi \cup \{\mathbf{sub}(\texttt{R}, \texttt{X})\}}$$

$$\text{(Ct-ThisInvk)}\quad \frac{\begin{array}{c}\Delta \vdash_{\text{co}} \texttt{this} : \chi \mid \langle \bullet \mid \bullet \rangle \mid \{\} \qquad \forall \texttt{e}_i \in \overline{\texttt{e}}, \quad \Delta \vdash_{\text{co}} \texttt{e}_i : \texttt{R}_i \mid \langle \overline{\texttt{F}}^{(i)} \mid \overline{\sigma}^{(i)} \rangle \mid \Phi_i \\ \texttt{X}_0 \ldots \texttt{X}_{\sharp(\overline{\texttt{e}})} \text{ fresh} \qquad \Delta(\texttt{this}) = \langle \cdots \mid \overline{\zeta} \rangle \qquad \mathit{choose}(\overline{\zeta}, \texttt{m}) = \texttt{N}_0\ \texttt{m}(\overline{\texttt{N}}\,\overline{\texttt{x}}) = \zeta \\ \Phi = (\cup_{i=1..\sharp(\overline{\texttt{e}})} \Phi_i) \cup \{\mathbf{vpa}(\chi, \texttt{this}, \texttt{N}_0, \texttt{X}_0)\} \cup (\cup_{i=1..\sharp(\overline{\texttt{e}})} \{\mathbf{vpa}(\chi, \texttt{this}, \texttt{N}_i, \texttt{X}_i), \mathbf{sub}(\texttt{R}_i, \texttt{X}_i)\})\end{array}}{\Delta \vdash_{\text{co}} \texttt{this.m}(\overline{\texttt{e}}) : \texttt{X}_0 \mid \langle \cup_{i \in 1..n} \overline{\texttt{F}}^{(i)} \mid (\cup_{i \in 1..n} \overline{\sigma}^{(i)}) \cup \zeta \rangle \mid \Phi}$$

$$\text{(CT-NonThisInvk)}\quad \frac{\begin{array}{c}\Delta \vdash_{\text{co}} \texttt{e}_0 : \texttt{O}_0 \mid \langle \overline{\texttt{F}}^{(0)} \mid \overline{\sigma}^{(0)} \rangle \mid \Phi_0 \\ \texttt{O}_0 \neq \chi \qquad \forall \texttt{e}_i \in \overline{\texttt{e}}, \quad \Delta \vdash_{\text{co}} \texttt{e}_i : \texttt{R}_i \mid \langle \overline{\texttt{F}}^{(i)} \mid \overline{\sigma}^{(i)} \rangle \mid \Phi_i \qquad \texttt{X}_0 \ldots \texttt{X}_{\sharp(\overline{\texttt{e}})} \texttt{Y}_0 \ldots \texttt{Y}_{\sharp(\overline{\texttt{e}})} \text{ fresh} \\ \Phi = (\cup_{i=0..\sharp(\texttt{e}_0\overline{\texttt{e}})} \Phi_i) \cup \{\mathbf{mTyp}(\texttt{O}_0, \texttt{m}, \texttt{X}_0 \ldots \texttt{X}_{\sharp(\overline{\texttt{e}})}), \mathbf{vpa}(\texttt{O}_0, \texttt{e}_0, \texttt{X}_0, \texttt{Y}_0)\} \cup (\cup_{i=1..\sharp(\overline{\texttt{e}})} \{\mathbf{vpa}(\texttt{O}_0, \texttt{e}_i, \texttt{X}_i, \texttt{Y}_i), \mathbf{sub}(\texttt{R}_i, \texttt{Y}_i)\})\end{array}}{\Delta \vdash_{\text{co}} \texttt{e}_0\texttt{.m}(\overline{\texttt{e}}) : \texttt{Y}_0 \mid \langle \cup_{i \in 1..n} \overline{\texttt{F}}^{(i)} \mid (\cup_{i \in 1..n} \overline{\sigma}^{(i)}) \rangle \mid \Phi}$$

$$\text{(CT-New)}\quad \frac{\cdots \texttt{class C}\langle\overline{\alpha}\rangle \texttt{ implements N} \cdots}{\Delta \vdash_{\text{co}} \texttt{new C}\langle\overline{\texttt{d}}\rangle : \texttt{N}[\overline{\texttt{d}}/\overline{\alpha}] \mid \langle \bullet \mid \bullet \rangle \mid \{\mathbf{isValid}(\texttt{C}\langle\overline{\texttt{d}}\rangle)\}}$$

$$\text{(CT-Cast)}\quad \frac{\Delta \vdash_{\text{co}} \texttt{e} : \texttt{R} \mid \langle \overline{\texttt{F}} \mid \overline{\sigma} \rangle \mid \Phi}{\Delta \vdash_{\text{co}} (\texttt{N})\texttt{e} : \texttt{N} \mid \langle \overline{\texttt{F}} \mid \overline{\sigma} \rangle \mid \Phi \cup \{\mathbf{isValid}(\texttt{N}), \mathbf{cast}(\texttt{N}, \texttt{R})\}}$$

Fig. 21. Expression constraint-based typing rules.

for `this`. The constraint that `this` must have a method `m` with signature ζ is collected in the conclusion of the rule, together with the constraints expressing that subtyping between actual and formal parameter types must hold, and the constraints expressing that the view-point adaptation of the parameters and return types of the method must have valid annotations. The rule for method invocation on a receiver different from `this`, (CT-NonThisInvk), is similar. The only difference is that a constraints expressing that the signature of `m` must be extracted from the type $\texttt{O}_0$ of the receiver is collected. Rules (CT-New) and (CT-Cast) are straightforward.

Rules (CT-Field), (Ct-ThisInvk), and (CT-NonThisInvk) are the only rules that create type variables. The type variable `X` created by rule (CT-Field) occur in the fourth argument of the constraint $\mathbf{vpa}(\chi, \texttt{this}, \texttt{N}, \texttt{X})$, the type variables $\texttt{X}_i$ created by rule (Ct-ThisInvk) occur in the fourth argument of the constraints $\mathbf{vpa}(\chi, \texttt{this}, \texttt{N}_i, \texttt{X}_i)$, and the type variables $\texttt{X}_i$ and $\texttt{Y}_i$ created by rule (CT-NonThisInvk) occur in the third argument of the constraint $\mathbf{mTyp}(\texttt{O}_0, \texttt{m}, \texttt{X}_0 \ldots \texttt{X}_{\sharp(\overline{\texttt{e}})})$ and in the fourth argument of the constraints $\mathbf{vpa}(\texttt{O}_0, \texttt{e}_i, \texttt{X}_i, \texttt{Y}_i)$, respectively. The checking rules for constraints (given in Section 5.2) can be applied by considering the constraints in the order in which they are created. In particular, when the constraints $\mathbf{vpa}(\cdot, \cdot, \cdot, \cdot)$ and $\mathbf{mTyp}(\cdot, \cdot, \cdot)$ are checked, their last arguments contain type variables only, and checking the constraints in a different order may not cause the instantiation of any of those type variables before the corresponding constraint is checked.

6. Relating ownership typing and constraint-based ownership typing

The following theorems state that the constraint-based ownership type system for IFBTJ ($\vdash_{\text{co}}$) satisfies the specification provided by the ownership type system for FIFBTJ ($\vdash_{\text{o}}$) and the conformance of the constraint-based ownership type system to the flattening principle, respectively.

Theorem 6.1 (*Equivalence of* $\vdash_{\text{co}}$*-typability and* $\vdash_{\text{o}}$*-typability on* FIFBTJ *Programs*)**.** *For every* FIFBTJ *program* $\texttt{P} = \overline{\texttt{ID}}\ \overline{\texttt{CD}}\ \texttt{e}$ *it holds that* $\vdash_{\text{co}} \texttt{P} : \texttt{N}$ *if and only if* $\vdash_{\text{o}} \texttt{P} : \texttt{N}$.

Proof. See Appendix A. □

ι			object identifiers
v	::=	$\iota \mid \texttt{null}$	values
b	::=	$\ldots \mid \iota$	extended domain owners
e	::=	$\ldots \mid \texttt{v}$	extended expressions
o	::=	$\langle \texttt{C}\langle\overline{\texttt{d}}\rangle, \overline{\texttt{f}} \mapsto \overline{\texttt{v}}\rangle$	objects
H	::=	$\overline{\iota} \mapsto \overline{\texttt{o}}$	heap

Fig. 22. Semantic entities used by the operational semantics.

Theorem 6.2 (*Flattening Preserves $\vdash_{\texttt{co}}$-typing*). *For every* IFBTJ *program* $\texttt{P} = \overline{\texttt{ID}}\ \overline{\texttt{TD}}\ \overline{\texttt{CD}}\ \texttt{e}$, *if* $\vdash_{\texttt{co}} \texttt{P} : \texttt{N}$ *then* $\vdash_{\texttt{co}} [\![\texttt{P}]\!] : \texttt{N}$.

Proof. See Appendix B. □

7. Operational semantics

In this section, we present the operational semantics of the flat language.

7.1. Semantic entities

In Fig. 22, the semantic entities that are used by the operational semantics are shown. Objects are expressed as a pair of the object type, $\texttt{C}\langle\overline{\texttt{d}}\rangle$, and a mapping from fields to their values, $\overline{\texttt{f}} \mapsto \overline{\texttt{v}}$. The crucial point here is that the domain parameters of the object type $\overline{\texttt{d}}$ are *runtime domains*, i.e., domains that have a *value* as owner. Besides `null`, a value can in particular be an object identifier ι. The owner of a runtime domain is always either `null` or is an instance of a box class. Its identifier can thus be used as a unique representation of the box at runtime. The first domain parameter ($\texttt{d}_1$) always represents the domain that owns the object. If the object is not of a box class, the box that the object belongs to is represented by the object that owns that domain. As runtime domains should be valid domains we extend the definition of *validOwner* with an additional case that allows object identifiers to be owners of domains:

$$\frac{}{\mathit{validOwner}(\iota)}$$

7.2. Evaluation contexts

An evaluation context $\texttt{e}_\Box$ is an expression with a "hole" □ somewhere inside [31]. We write $\texttt{e}_\Box[\texttt{e}']$ to fill the hole of e with the expression $\texttt{e}'$. The syntax of evaluation contexts is as follows:

$$\texttt{e}_\Box ::= \Box \mid \texttt{this.f} = \texttt{e}_\Box \mid \texttt{e}_\Box\texttt{.m}(\overline{\texttt{e}}) \mid \texttt{v.m}(\overline{\texttt{v}}, \texttt{e}_\Box, \overline{\texttt{e}}) \mid (\texttt{N})\texttt{e}_\Box$$

7.3. Reduction rules

We use a big-step operational semantics with rules of the form

$$\texttt{H};\ \texttt{e} \Downarrow_{\texttt{v}'} \texttt{H}';\ \texttt{v}$$

meaning that expression e under heap H is reduced to v and new heap $\texttt{H}'$, where $\texttt{v}'$ is the current context object, or `null`, if the execution takes place in the global context. The rules that define the reduction relation are shown in Fig. 23.

To update the value of a field f in object o to value v, we write $\texttt{o}[\texttt{f} \mapsto \texttt{v}]$, which returns the updated object, and $\texttt{o}(\texttt{f})$, which returns the value of field f. The object type is given by *type*(o).

Method invocation is handled by rule (R-Call). The rule replaces the formal parameters of the method body by the actual parameters, adapts `this` to the receiving object, and substitutes the keyword `box` by the owning box of the receiving object. This is the runtime correspondent to the viewpoint adaptation of the type system.

All other rules are standard.

7.4. Program execution

A program $\texttt{P} = \overline{\texttt{ID}}\ \overline{\texttt{CD}}\ \texttt{e}$ in FIFBTJ is executed by evaluating e under an empty heap and context object `null`:

$$\varnothing;\ \texttt{e} \Downarrow_{\texttt{null}} \texttt{H};\ \texttt{v}$$

In our semantics we have `null` as the owner of the global domain, and so we start the execution with a configuration that only maps `null` to the global domain, and everything else is empty.

$$\frac{\mathtt{v} = \mathtt{H}(\iota)(\mathtt{f})}{\mathtt{H};\, \iota.\mathtt{f} \Downarrow_{\iota} \mathtt{H};\, \mathtt{v}}\ (\textsc{r-field-read}) \qquad \frac{\mathtt{o} = \mathtt{H}(\iota)[\mathtt{f} \mapsto \mathtt{v}] \qquad \mathtt{H}' = \mathtt{H}[\iota \mapsto \mathtt{o}]}{\mathtt{H};\, \iota.\mathtt{f} = \mathtt{v} \Downarrow_{\iota} \mathtt{H}';\, \mathtt{v}}\ (\textsc{r-field-update})$$

$$\frac{\begin{array}{c}\mathtt{G} = \mathit{type}(\mathtt{H}(\iota)) \qquad \mathit{mbody}(\mathtt{G}, \mathtt{m}) = (\overline{\mathtt{x}}, \mathtt{e}) \\ \iota'' = \mathit{boxOwner}(\mathtt{G}, \iota) \qquad \mathtt{e}' = \mathtt{e}[\iota''/\mathtt{box}, \iota/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}] \qquad \mathtt{H};\, \mathtt{e}' \Downarrow_{\iota} \mathtt{H}';\, \mathtt{v}\end{array}}{\mathtt{H};\, \iota.\mathtt{m}(\overline{\mathtt{v}}) \Downarrow_{\mathtt{v}'} \mathtt{H}';\, \mathtt{v}}\ (\textsc{r-call})$$

$$\frac{\iota \notin \mathit{dom}(\mathtt{H}) \qquad \mathtt{H}' = \mathtt{H}[\iota \mapsto \langle \mathtt{C}\langle\overline{\mathtt{d}}\rangle, \overline{\mathtt{f}} \mapsto \overline{\mathtt{null}}\rangle]}{\mathtt{H};\, \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{d}}\rangle \Downarrow_{\mathtt{v}'} \mathtt{H}';\, \iota}\ (\textsc{r-new-object}) \qquad \frac{}{\mathtt{H};\, (\mathtt{N})\mathtt{null} \Downarrow_{\mathtt{v}'} \mathtt{H};\, \mathtt{null}}\ (\textsc{r-cast-null}) \qquad \frac{\mathit{type}(\mathtt{H}(\iota)) <: \mathtt{N}}{\mathtt{H};\, (\mathtt{N})\iota \Downarrow_{\mathtt{v}'} \mathtt{H};\, \iota}\ (\textsc{r-cast})$$

$$\frac{\mathtt{H};\, \mathtt{e} \Downarrow_{\mathtt{v}'} \mathtt{H}';\, \mathtt{v} \qquad \mathtt{H}';\, \mathtt{e}_{\square}[\mathtt{v}] \Downarrow_{\mathtt{v}'} \mathtt{H}'';\, \mathtt{v}''}{\mathtt{H};\, \mathtt{e}_{\square}[\mathtt{e}] \Downarrow_{\mathtt{v}'} \mathtt{H}'';\, \mathtt{v}''}\ (\textsc{r-context})$$

Fig. 23. Rules to evaluate expressions.

8. Soundness of the ownership type system

This section shows the soundness of the ownership type system. Soundness is proved by a standard subject reduction theorem, which states that, if an expression is correctly typed in the ownership type system, then a value that is reduced from this expression is a subtype of the type of the expression. In addition, this theorem then proves the encapsulation property of the ownership type system, namely that all values that can appear in the context of an object are accessible by that object.

8.1. Heap typing

The type system presented in Section 4 is only defined on *source expressions*, i.e., expressions that do not contain object identifiers. In order to type object identifiers, we introduce a heap typing Θ, which assigns types to object identifiers.

$$\Theta ::= \overline{\iota} \mapsto \overline{\mathtt{G}}$$

To have a concise notation to type values in general, we also define $\Theta(\mathtt{null}) \stackrel{\text{def}}{=} \bot$.

We also define a notion of extending heap typings as follows.

Definition 8.1 (*Heap Typing Extension*)**.**

$$\Theta \subseteq \Theta' \stackrel{\text{def}}{=} \forall \iota \in \mathit{dom}(\Theta) : \Theta(\iota) = \Theta'(\iota)$$

8.2. Extended type rules

For the type soundness proof we have to extend the type rules for typing expressions with a heap typing and a context value $\mathtt{v}$, which represents the current context object, in which the expression is typed: $\Theta;\, \mathtt{v};\, \Gamma \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{G}$. The Θ parameter is required to be able to type object identifiers, and the $\mathtt{v}$ parameter is required to be able to verify the correctness of runtime domains. The definition of this judgment is equal to the definition given in Fig. 17, where the additional parameters Θ and $\mathtt{v}$ are just ignored by the rules, but passed to type judgments of preconditions. The rules in Fig. 17 remain the same, except that rules (T-METHOD) and (T-PROG) type the expression $\mathtt{e}$, i.e., the body expression (respectively, the main expression) under the empty heap typing $\varnothing$ and $\mathtt{null}$ as context object. Note that the heap typing thus has no influence on the typing of *source* programs.

A single rule is added to type object identifiers:

$$\frac{\Theta;\, \mathtt{v};\, \Gamma \vdash_{\mathtt{or}} \Theta(\iota)}{\Theta;\, \mathtt{v};\, \Gamma \vdash_{\mathtt{ox}} \iota : \Theta(\iota)}\ (\textsc{t-oid})$$

$$\frac{}{\Theta \vdash_{\mathtt{or}} \iota.\mathtt{c} \rightarrow_{\mathtt{r}} \iota.\mathtt{c}'}\ (\text{AR-BOX}) \qquad \frac{}{\Theta \vdash_{\mathtt{or}} \mathtt{b.c} \rightarrow_{\mathtt{r}} \mathtt{null.c}}\ (\text{AR-NULL}) \qquad \frac{\Theta \vdash_{\mathtt{or}} \iota.\mathtt{c} \rightarrow_{\mathtt{r}} \mathit{odom}(\Theta(\iota'))}{\Theta \vdash_{\mathtt{or}} \iota.\mathtt{c} \rightarrow_{\mathtt{r}} \iota'.\mathtt{boundary}}\ (\text{AR-BOUNDARY})$$

$$\frac{^{2}\mathtt{C}\langle\overline{\mathtt{d}}\rangle = \Theta(\iota') \qquad \mathit{boxOwner}(\Theta(\iota'), \iota') = \iota}{\Theta \vdash_{\mathtt{or}} \iota.\mathtt{c} \rightarrow_{\mathtt{r}} \mathtt{d}_i}\ (\text{AR-PARAM})$$

Fig. 24. Definition of the accessibility relation on runtime domains.

In addition, we also have to extend the set of valid domains, because domains can now be owned by object identifiers. The only restriction is that owners of domains can only be instances of box classes.

$$\frac{\mathtt{v} = \mathtt{null} \vee \mathit{isBoxType}(\Theta(\mathtt{v}))}{\Theta \vdash_{\mathtt{or}} \mathtt{v.c}}\ (\text{V-DOMAIN-OID})$$

Finally, we have to change the typing of valid types, to allow types with object identifiers as owners. Note, the context object is `null` when checking source programs, thus the typing is not changed.

$$\frac{}{\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{or}} \bot}\ (\text{V-TYPE-NULL})$$

$$\frac{\Theta \vdash_{\mathtt{or}} \overline{\mathtt{d}} \qquad \Theta \vdash_{\mathtt{or}} \mathtt{d}_1 \rightarrow_{\mathtt{r}} \overline{\mathtt{d}} \qquad \mathtt{v}' = \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v}) \qquad \Theta \vdash_{\mathtt{or}} \mathtt{v}'.\mathtt{c} \rightarrow_{\mathtt{r}} \mathtt{d}_1 \qquad |\mathit{params}(\mathtt{U})| = |\overline{\mathtt{d}}|}{\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{or}} \mathtt{U}\langle\overline{\mathtt{d}}\rangle}\ (\text{V-TYPE-OID})$$

The accessibility relation now has to be defined on domains with ι as owners. The relation is denoted by $\mathtt{d} \rightarrow_{\mathtt{r}} \mathtt{d}'$ and shown in Fig. 24. The rules are much simpler than the accessibility rules defined in Fig. 14. They essentially capture the encapsulation invariant that is provided by the type system namely an object ι can access a domain iff

1. the domain is owned by the owner of ι;
2. it is the global domain, i.e., a domain owned by `null`; and
3. the domain is a boundary domain owned by an object that is accessible.

8.3. Type-correct heaps

Given a heap typing Θ, we define a correctly typed heap under heap typing Θ.

$$\frac{\begin{array}{c}\forall \iota \in \mathit{dom}(\mathtt{H}) : \\ \mathtt{G} = \Theta(\iota) \qquad \Theta; \iota; \varnothing \vdash_{\mathtt{or}} \mathtt{G}\end{array} \qquad \mathtt{H}(\iota) = \langle \mathtt{G}, \overline{f} \mapsto \overline{\mathtt{v}} \rangle \qquad \overline{\mathtt{G}} = \mathit{field}(\mathtt{G}, \overline{\mathtt{f}}) \qquad \mathtt{G}; \mathtt{G}; \iota \vdash_{\mathtt{o}} \overline{\mathtt{G}} \triangleright \overline{\mathtt{G}}'' \qquad \Theta; \iota; \mathtt{this} : \mathtt{G} \vdash_{\mathtt{ox}} \overline{\mathtt{v}} : \overline{\mathtt{G}}' \qquad \overline{\mathtt{G}}' <: \overline{\mathtt{G}}''}{\Theta \vdash_{\mathtt{or}} \mathtt{H}}\ (\text{T-HEAP})$$

8.4. Properties of the ownership type system

We prove type-soundness of the ownership type system by the standard subject reduction proof, which states that, when a well-typed expression can be reduced to a value, the type of the value is a subtype of the original expression. For the proof invariant it is also needed to show that the heap stays well-typed under a possibly extended heap typing.

Theorem 8.2 (*Subject Reduction*). *Assume* $\vdash_{\mathtt{ox}} \mathtt{P}$, $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, $\Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_{\mathtt{t}} \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{G}_{\mathtt{e}}$, $\mathtt{v}' = \mathtt{null} \vee \Theta(\mathtt{v}') = \mathtt{G}_{\mathtt{t}}$, *and* $\mathtt{this} \notin \mathtt{e}$. *If* $\mathtt{H}; \mathtt{e} \Downarrow_{\mathtt{v}'} \mathtt{H}'; \mathtt{v}$, *then there exists a* Θ' *with* $\Theta \subseteq \Theta'$, $\Theta'; \mathtt{v}'; \mathtt{this} : \mathtt{G}_{\mathtt{t}} \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{G}_{\mathtt{v}}$, $\mathtt{G}_{\mathtt{v}} <: \mathtt{G}_{\mathtt{e}}$, *and* $\Theta' \vdash_{\mathtt{ox}} \mathtt{H}'$.

Proof. See Appendix C. □

From the subject reduction theorem, it is straightforward to derive the encapsulation invariant that states that local objects of a box are encapsulated, i.e., cannot be accessed by the environment of the box. Technically speaking, this means that at all object identifiers that are reducible in the context of a certain object, are accessible by that object; i.e., the owner domain is accessible.

Theorem 8.3 (*Encapsulation Invariant*). *Assume* $\vdash_{\mathtt{o}} \mathtt{P}$, $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, $\Theta; \mathtt{v}; \mathtt{this} : \mathtt{G}_{\mathtt{t}} \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{G}_{\mathtt{e}}$, $\mathtt{v} = \mathtt{null} \vee \Theta(\mathtt{v}) = \mathtt{G}_{\mathtt{t}}$, $\mathtt{this} \notin \mathtt{e}$. *If* $\mathtt{H}; \mathtt{e} \Downarrow_{\mathtt{v}} \mathtt{H}'; \iota$, *then* $\Theta \vdash_{\mathtt{or}} \mathit{boxOwner}(\mathtt{H}, \mathtt{v}).c \rightarrow_{\mathtt{r}} \mathit{odom}(\mathtt{H}'(\iota))$.

9. Related work

The literature on traits and boxes has been partially quoted throughout the paper. Here, we briefly discuss the relation with other type systems for traits and with other ownership type systems.

Various type systems for traits have been proposed in the literature [59,32,63,12,60,13,43,42]. These approaches guarantee that the composed program is type correct; i.e., all required fields and methods are present with the appropriate types. The IFBTJ constraint-based ownership type system builds on the constraint-based type system of the FRJ calculus [13]. Both the IFBTJ and the FRJ type systems support the type-checking of traits in isolation from the classes or traits that use them, so that it is possible to type-check a method defined in a trait only once (instead of having to type-check it in every class using that trait). A distinguishing feature of both IFBTJ and FRJ with respect to the other formulations of traits within a JAVA-like nominal type system enjoying this property [63,60,42,6] is that the method exclusion and method/field renaming operations are fully supported (as in the formulation of traits in a structurally typed setting given by Reppy and Turon [59]). We are not aware of any previous proposal of an ownership type system for a trait-based language.

The basic idea of the box component model, namely to hierarchically structure the heap into dynamically created regions, originated from the notion of ownership types. Ownership types are a static way to guarantee encapsulation of objects. The notion of ownership types stems from Clarke [24,23,21] as an approach to formalize the core of Flexible Alias Protection [53]. Ever since, many researchers have investigated ownership type systems [49,5,16,58,27, to name a few]. Ownership type systems have been used to prevent data-races [17], deadlocks [15,14], and to allow the modular specification and verification of object-oriented programs [48,29].

Most ownership type systems guarantee the so-called *owners-as-dominators* property, which states that all accesses from external objects to owned objects must go through the owner object. This property does not allow for multiple objects at the boundary of a component. Several variations and extensions of the pure ownership type approach have been proposed. Clarke and Drossopoulou [22] weakened the owners-as-dominators property by allowing dynamic aliases, i.e., aliases stored on the stack, to access owned objects. Other approaches [21,15] allow JAVA's inner member classes [34] to access the representation objects of their parent objects. Both solutions do not provide the full power to generalize the owners-as-dominators property. *Ownership Domains* (OD) [3] was the first approach to fully support multi-access ownership contexts. Objects are not owned directly by other objects, but are owned by domains, which are in turn owned by objects. Every object can have an arbitrary number of domains, which can either be *private* or *public*. Objects in the private domain are encapsulated, and objects in the public domain can be accessed by the outside. Programmers define which domains can access which other domains by *link* declarations. OD have been combined with an effects system [64]. A more general version of OD has been formalized in System F [37]. An extension of the ownership approach is MOJO [19], which allows multiple owners per object, thus not restricting the ownership structure to a tree anymore.

The simplification of having only two domains for each object, namely a local and a boundary one, was introduced by *Simple Loose Ownership Domains* [61]. This approach also allows for abstracting from the concrete owner of a domain by introducing the notion of *loose* domains, a feature which could be incorporated into our type system as well. The first approach to apply simple ownership domains to boxes has been presented in [55]. This approach was formalized in a standard object-oriented language without class-inheritance and supported the inference of ownership annotations in the context of modules.

Lu and Potter [45] presented a type system which separates object ownership and accessibility. Instead of only giving the owner of a type, types are also annotated by their possible accessibility. This introduces a very flexible system, which allows programming patterns not possible with our ownership type system. However, this flexibility comes with the cost of a higher annotation overhead, as both the owner and the accessibility must be given.

A previous system [46] considers the encapsulation of effects instead of objects. This makes it possible to access internal representation objects from the outside, but disallows their direct modification. This mechanism is similar to the read-only mechanism of the Universes approach [49,27], where it is allowed to have read-only references to representation objects. However, this approach forbids programming patterns where boundary objects should be able to directly change the state of representation objects without using the owner object.

The basic idea of components as dynamic entities is also used in ArchJava [4]. Like boxes, ArchJava's components are dynamic (hierarchical) entities with a well-defined interface. In contrast to the box model, the interface is comprised of ports describing (required and provided) methods. ArchJava has been combined with alias protection [2] to be able to protect the data passing along the connections between ports. But as ArchJava focuses on expressing the system architecture explicitly in the code, the language produces much more overhead than the box model.

In ownership type systems, the annotations may hide the ownership structure from the programmer. Recently, tribal ownership [20] proposes using the nested class structure of a program as the ownership structure. This makes the ownership structure explicitly visible and leads to an annotation-free ownership system. But because encapsulation of objects is bound to the visibility of the object's class, it is less flexible than other ownership type systems.

Confined Types [67,68] is a lightweight mechanism that allows the encapsulation of objects within the boundary of a JAVA package. Confined types can be relatively easily inferred from unannotated programs [35].

In order to lower the annotation burden for the programmer, several inference algorithms for unannotated (e.g. [5,28, 47]) or partially annotated code ([55,1,44] and others) have been developed.

Beside type systems, there are other possibilities to statically ensure object encapsulation, for example ,by using general specification and verification frameworks like Spec# and JML [40,29].

10. Conclusions and future work

Encapsulation and reuse are two orthogonal but not unrelated concepts of programming languages. In general, encapsulation tends to hinder reuse, and vice versa. The box model is a lightweight component model for the object-oriented paradigm, which structures the flat object-heap into hierarchical runtime components called boxes. Static encapsulation of objects can be achieved by ownership type systems. Up to now, the box model has been presented in class-based languages without inheritance. Traits are a mechanism for code reuse that offers more flexibility than standard class-based inheritance. In this paper, we have presented a combination of traits with an ownership type system for boxes in a Java-like setting and formalized it by means of a minimal core calculus. This combination also solves a specific problem of the box ownership type system. Namely, although the box model could be extended to languages with single inheritance (as done for other ownership type systems), box classes could not inherit from standard classes (and vice versa), and thus code sharing between these two types of classes would not be possible.

To the best of our knowledge, ownership type systems have been presented only in the setting of class-based languages (possibly) with inheritance. So, the proposal described in this paper represents the first attempt to define an ownership type system for a language with traits. Each trait definition is type-checked in isolation from the classes and traits that use it. When analyzing a trait definition, type constraints that include the necessary ownership checks are generated. When classes are assembled by composing traits, the type system ensures that the ownership conditions are not violated. We believe that this approach, which builds on the technique for type-checking traits within a nominal Java-like type system proposed in [13], could be used to integrate traits within other ownership type systems.

In future work we would like to develop a prototypical implementation of a programming language based on the IFBTJ calculus. We are also planning to extend the IFBTJ type system to deal with generics. Another possible direction for future work would be to add ownership transfer to our type system. By using the notion of external uniqueness and applying the techniques described for example in [50,25], we could gain flexibility about the contents of a box and be able to support the factory pattern and other ownership-unfriendly patterns [51].

Acknowledgement

We are grateful to Marco De Luca for our initial collaboration on the subject of this paper. We thank the anonymous SCP referees, the anonymous Coordination 2010 referees, Dave Clarke, and Susan Eisenbach for insightful comments and suggestions for improving the presentation.

Appendix A. Proof of Theorem 6.1

In order to simplify the presentation of the proof, we introduce a variant of system $\vdash_{\mathtt{co}}$, denoted by $\vdash'_{\mathtt{co}}$, that is customized for FIFBTJ programs. The rules of system $\vdash'_{\mathtt{co}}$ are obtained from the rules of system $\vdash_{\mathtt{co}}$ by

- modifying rule (CT-Program) in Fig. 19 by dropping the premise $\vdash_{\mathtt{co}} \overline{\mathtt{TD}} : \cdots$ and by removing $\overline{\mathtt{TD}}$ from the conclusion;
- dropping rules (CT-Trait) and (CT-TEBasic) in Fig. 19 and all the rules in Fig. 20;
- replacing rule (CT-Class) in Fig. 19 by rule (CT-Class′) in Fig. 25.

The following lemma states that $\vdash_{\mathtt{co}}$ and $\vdash'_{\mathtt{co}}$ are equivalent on FIFBTJ programs.

Lemma A.1 (*Equivalence of* $\vdash_{\mathtt{co}}$*-typability and* $\vdash'_{\mathtt{co}}$*-typability on* FIFBTJ *Programs*)**.** *For every* FIFBTJ *program* $\mathtt{P} = \overline{\mathtt{ID}}\ \overline{\mathtt{CD}}\ \mathtt{e}$ *it holds that* $\vdash_{\mathtt{co}} \mathtt{P} : \mathtt{N}$ *if and only if* $\vdash'_{\mathtt{co}} \mathtt{P} : \mathtt{N}$.

Proof. Straightforward. □

In order to be able to relate the open type and the type inferred for an expression by $\vdash'_{\mathtt{co}}$ and $\vdash_{\mathtt{o}}$, respectively, we extend the constraints and the constraint satisfaction checking rules given in Fig. 18 by adding the constraint $\mathbf{eq}(\mathtt{L}, \mathtt{R})$ and the rule

$$\frac{\Gamma \vdash_{\mathtt{co}} \Phi}{\Gamma \vdash_{\mathtt{co}} \Phi \uplus \{\mathbf{eq}(\mathtt{L}, \mathtt{L})\}}\ (\text{CC-Equals})$$

Lemma A.2. *For every class definition* $\ldots$ `class` $\mathtt{C}\langle\overline{\alpha}\rangle$ `implements` $\mathtt{I}\langle\overline{\mathtt{d}}\rangle$ `by` $\{\overline{\mathtt{F}};\ \bullet;\ \overline{\mathtt{M}}\}$ $\{\ \overline{\mathtt{N}}\ \overline{\mathtt{f}};\ \}$*, it holds that* $\mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle,\ \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash_{\mathtt{o}} \mathtt{e} : \mathtt{L}$ *if and only if*

1. $\mathtt{this} : \langle\ \overline{\mathtt{F}}\ \triangleright\ \mathit{mSig}(\overline{\mathtt{M}})\ \rangle,\ \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash'_{\mathtt{co}} \mathtt{e} : \mathtt{R} \triangleright \langle\ \overline{\mathtt{F}'}\ \triangleright\ \overline{\sigma'}\ \rangle \triangleright \Phi$*, and*
2. $\mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle,\ \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash'_{\mathtt{co}} (\Phi \cup \{\mathbf{eq}(\mathtt{L}, \mathtt{R})\})[{}^{\mathtt{C}\langle\overline{\alpha}\rangle}/_{\chi}]$.

$$\frac{\begin{array}{c}\overline{\mathtt{M}} = \mathtt{M}_1 \ldots \mathtt{M}_p \qquad p \geq 0 \qquad \mathit{annVars}(\{\,\overline{\mathtt{F}};\ \bullet;\ \overline{\mathtt{M}}\,\}) \subseteq \overline{\alpha} \qquad \mathit{mSig}(\overline{\mathtt{M}}) = \overline{\zeta} = \zeta_1 \ldots \zeta_p \\ \forall i \in 1..p, \qquad \langle\, \overline{\mathtt{F}} \mid \overline{\zeta} \,\rangle \vdash_{\mathtt{co}} \mathtt{M}_i : \mu_i \qquad \mu_i = \zeta_i \mid \langle\, \overline{\mathtt{F}}^{(i)} \mid \overline{\sigma}^{(i)} \,\rangle \mid (\overline{\mathtt{x}}^{(i)}, \Phi_i) \\ \zeta_i = \mathtt{N}_i\ (\overline{\mathtt{N}}^{(i)}) \qquad \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle,\ \overline{\mathtt{x}}^{(i)} : \overline{\mathtt{N}}^{(i)} \vdash_{\mathtt{co}} \Phi_i[{}^{\mathtt{C}\langle\overline{\alpha}\rangle}/_{\chi}] \\ \cup_{i \in 1..p} \overline{\sigma}^{(i)} \subseteq \zeta \qquad (\cup_{i \in 1..p} \overline{\mathtt{F}}^{(i)}) = \overline{\mathtt{N}}\ \overline{\mathtt{f}} \qquad \mathit{mSig}(\mathtt{I}\langle\overline{\mathtt{d}}\rangle) \subseteq \zeta_1 \ldots \zeta_p \\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle \vdash_{\mathtt{o}} \mathtt{I}\langle\overline{\mathtt{d}}\rangle \qquad \alpha_1 = \mathtt{d}_1 \qquad \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle \vdash_{\mathtt{o}} \overline{\mathtt{N}} \qquad \mathit{isBoxType}(\mathtt{C}) \Leftrightarrow \mathit{isBoxType}(\mathtt{I}) \end{array}}{\vdash_{\mathtt{co}} \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{I}\langle\overline{\mathtt{d}}\rangle\ \mathtt{by}\ \{\,\overline{\mathtt{F}};\ \bullet;\ \overline{\mathtt{M}}\,\}\ \{\,\overline{\mathtt{N}}\ \overline{\mathtt{f}};\ \}}$$

(CT-Class′)

Fig. 25. Flat class constraint-based typing rule.

Proof. By structural induction on typing derivations, using the constraint satisfaction checking rules. We sketch the cases for the "only if" direction. The cases for the "if" direction are similar.

Case (T-NULL). By rules (CT-Null), (CC-Empty), and (CC-Equals).
Case (T-VAR). By rules (CT-Var), (CC-Empty), and (CC-Equals).
Case (T-FIELD). By rules (CT-Field), (CC-vpa), and (CC-Equals).
Case (T-FIELD-UP). By rules (CT-FieldUp), (CC-sub), and (CC-Equals).
Case (T-INVK). By rules (Ct-ThisInvk), (CC-vpa), (CC-sub), and (CC-Equals), if $\mathtt{e} = \mathtt{this.m}(\cdots)$. By rules (CT-NonThisInvk), (CC-mTyp), (CC-vpa), (CC-sub), and (CC-Equals), otherwise.
Case (T-NEW-CLASS). By rules (CT-New), (CC-isValid), and (CC-Equals).
Case (T-CAST). By rules (CT-Cast), (CC-isValid), (CC-cast), and (CC-Equals). □

Lemma A.3. *For every class definition* $\ldots \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{I}\langle\overline{\mathtt{d}}\rangle\ \mathtt{by}\ \{\,\overline{\mathtt{F}};\ \bullet;\ \overline{\mathtt{M}}\,\}\ \{\,\overline{\mathtt{N}}\ \overline{\mathtt{f}};\ \}$ *and method definition* $\mathtt{M} \in \overline{\mathtt{M}}$*, it holds that* $\mathtt{C}\langle\overline{\alpha}\rangle \vdash_{\mathtt{o}} \mathtt{M}$ *if and only if*

1. $\langle\, \overline{\mathtt{F}} \mid \mathit{mSig}(\overline{\mathtt{M}}) \,\rangle \vdash'_{\mathtt{co}} \mathtt{M} : \mathit{mSig}(\mathtt{M}) \mid \langle\, \overline{\mathtt{F}}' \mid \overline{\sigma}' \,\rangle \mid (\overline{\mathtt{x}}, \Phi)$*, and*
2. $\mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle,\ \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash'_{\mathtt{co}} \Phi[{}^{\mathtt{C}\langle\overline{\alpha}\rangle}/_{\chi}]$.

Proof. By Lemma A.2. □

Lemma A.4. *For every class definition* $\mathtt{CD} = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathtt{implements}\ \mathtt{I}\langle\overline{\mathtt{d}}\rangle\ \mathtt{by}\ \{\,\overline{\mathtt{F}};\ \bullet;\ \overline{\mathtt{M}}\,\}\ \{\,\overline{\mathtt{N}}\ \overline{\mathtt{f}};\ \}$*, it holds that* $\vdash_{\mathtt{o}} \mathtt{CD}$ *if and only if* $\vdash'_{\mathtt{co}} \mathtt{CD}$

Proof. By Lemma A.3. □

Proof of Theorem 6.1 (*Equivalence of* $\vdash_{\mathtt{co}}$*-typability and* $\vdash_{\mathtt{o}}$*-typability on* FIFBTJ *Programs*)**.** Straightforward, by Lemmas A.1, A.2 and A.4. □

Appendix B. Proof of Theorem 6.2

The sequence of the field names and the sequence of the method names selected on `this` in the expressions `e` are denoted by *fN*(`e`) and *mN*(`e`), respectively. The sequence of the field names and the sequence of the method names selected on `this` in the method declaration $\mathtt{M} = \mathtt{N\ m}\ (\overline{\mathtt{N}}\ \mathtt{x})\{\mathtt{return\ e}\}$ are given by $\mathit{fN}(\mathtt{M}) = \mathit{fN}(\mathtt{e})$ and $\mathit{mN}(\mathtt{M}) = \mathit{mN}(\mathtt{e})$. The definitions of *fN* and *mN* naturally extend to sequences of expressions and sequences of method definitions.

Recall that the flattening $[\![\mathtt{TE}]\!]$ of a trait expression TE yields a sequence of methods (see Section 3.2). A sequence of methods $\overline{\mathtt{M}}$ is well-typed if and only if all methods in $\overline{\mathtt{M}}$ are well-typed. In the following, we will write "$\langle\, \overline{\mathtt{F}} \mid \overline{\sigma} \,\rangle \vdash_{\mathtt{co}} \mathtt{M}_1 \ldots \mathtt{M}_n : \mu_1 \ldots \mu_n$" as short for "$\langle\, \overline{\mathtt{F}} \mid \overline{\sigma} \,\rangle \vdash_{\mathtt{co}} \mathtt{M}_1 : \mu_1, \ldots, \langle\, \overline{\mathtt{F}} \mid \overline{\sigma} \,\rangle \vdash_{\mathtt{co}} \mathtt{M}_n : \mu_n$".

Lemma B.1. *If* $\langle\, \overline{\mathtt{F}} \mid \overline{\sigma} \,\rangle \vdash_{\mathtt{co}} \mathtt{N\ m}\ (\overline{\mathtt{N}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\} : \mathtt{N}_0\ \mathtt{m}(\overline{\mathtt{N}}) \mid \langle\, \overline{\mathtt{F}}' \mid \overline{\sigma}' \,\rangle \mid (\overline{\mathtt{x}}, \Phi)$*, then* $\langle\, \overline{\mathtt{F}}'' \mid \overline{\sigma}'' \,\rangle \vdash_{\mathtt{co}} \mathtt{N\ m}\ (\overline{\mathtt{N}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e;}\} : \mathtt{N}_0\ \mathtt{m}(\overline{\mathtt{N}}) \mid \langle\, \overline{\mathtt{F}}' \mid \overline{\sigma}' \,\rangle \mid (\overline{\mathtt{x}}, \Phi)$ *for all* $\overline{\mathtt{F}}'' \supseteq \overline{\mathtt{F}}'$ *and* $\overline{\sigma}'' \supseteq \overline{\sigma}'$.

Proof. By structural induction on typing derivations. □

Lemma B.2. *If* $\alpha_1 \vdash_{\mathtt{co}} \mathtt{TE} : \overline{\mu}$*,* $\mathit{annVars}(\mathtt{TE}) \subseteq \alpha_1, \ldots = \overline{\alpha}$ *and* $\mathit{progVars}(\overline{\mathtt{d}}) = \bullet$*, then* $\vdash_{\mathtt{co}} \mathtt{TE}[{}^{\overline{\mathtt{d}}}/_{\overline{\alpha}}] : \overline{\mu}[{}^{\overline{\mathtt{d}}}/_{\overline{\alpha}}]$.

Proof. By structural induction on typing derivations. □

Lemma B.3. *Let* $\beta \vdash_{\mathtt{co}} \mathtt{TE} : \mu_1 \ldots \mu_n$*, where* $\mu_i = \zeta_i \mid \langle\, \overline{\mathtt{F}}^{(i)} \mid \overline{\sigma}^{(i)} \,\rangle$*. Then* $\langle\, \overline{\mathtt{F}} \mid \overline{\zeta} \,\rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}]\!] : \mu_1 \ldots \mu_n$*, where* $\overline{\mathtt{F}} = \overline{\mathtt{F}}^{(1)} \cup \ldots \cup \overline{\mathtt{F}}^{(n)}$ *and* $\overline{\zeta} = \zeta_1 \ldots \zeta_n \cup \overline{\sigma}^{(1)} \cup \ldots \cup \overline{\sigma}^{(n)}$.

Proof. By case induction on the flattening translation for trait expressions defined in Fig. 7.

Case $[\![\{\,\overline{\mathtt{F}};\ \overline{\mathtt{S}};\ \overline{\mathtt{M}}\,\}]\!]$**.** This is the base case of the induction. Straightforward, by rule (CT-TEBasic) in Fig. 19.

Case $[\![\mathtt{T}\langle\overline{\mathtt{d}}\rangle]\!]$**.** Straightforward, by induction, using Lemma B.2.

Case $[\![\mathtt{TE}_1 + \mathtt{TE}_2]\!]$**.** By induction, we have that $\langle\ \overline{\mathtt{F}}'\ \mid\ \overline{\zeta}'\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}_1]\!] : \overline{\mu}'$ and $\langle\ \overline{\mathtt{F}}''\ \mid\ \overline{\zeta}''\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}_2]\!] : \overline{\mu}''$. The result follows by Lemma B.1.

Case $[\![\mathtt{TE}\,[\mathtt{exclude\ m}]]\!]$**.** By induction, we have that $\langle\ \overline{\mathtt{F}}'\ \mid\ \overline{\zeta}'\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}]\!] : \overline{\mu}$. Then we have two possible cases for the type of `this` in the typing of the sequence of methods $[\![\mathtt{TE}[\mathtt{exclude\ m}]]\!]$: (i) $\mathtt{m} \notin mN([\![\mathtt{TE}[\mathtt{exclude\ m}]]\!])$. Then, for each method $\mathtt{n} \neq \mathtt{m}$ in *names*$([\![\mathtt{TE}\,[\mathtt{exclude\ m}]]\!])$, we have that $\mathtt{this} : \langle\ \overline{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m})\ \rangle \vdash_{\mathtt{co}} \mathtt{I\ n}\ (\overline{\mathtt{I}}\ \overline{\mathtt{x}})\ \{\mathtt{return\ e;}\} : \mu$, where $\mathtt{I\ n}\ (\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\} = \mathit{choose}([\![\mathtt{TE}\,[\mathtt{exclude\ m}]]\!], \mathtt{n})$ and $\mu = \mathit{choose}(\overline{\mu}, \mathtt{n})$, by Lemma B.1. So $\mathtt{this} : \langle\ \overline{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m})\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}\,[\mathtt{exclude\ m}]]\!] : \mathit{exclude}(\overline{\mu}, \mathtt{m})$. (ii) $\mathtt{m} \in mN([\![\mathtt{TE}\,[\mathtt{exclude\ m}]]\!])$. Then we have that $\mathtt{this} : \langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}\,[\mathtt{exclude\ m}]]\!] : \mathit{exclude}(\overline{\mu}, \mathtt{m})$.

Case $[\![\mathtt{TE}\,[\mathtt{m\ aliasAs\ m}']]\!]$**.** This case is similar to the case $[\![\mathtt{TE}_1 + \mathtt{TE}_2]\!]$.

Case $[\![\mathtt{TE}\,[\mathtt{rename\ m\ to\ m}']]\!]$**.** By induction, we have that $\mathtt{this} : \langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}]\!] : \overline{\mu}$. Then we have two possible cases for the type of `this` in the typing of the sequence of methods $mR([\![\mathtt{TE}]\!], \mathtt{m}, \mathtt{m}')$: (i) $\langle\ \overline{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m}) \cup (\mathit{choose}(\overline{\sigma}, \mathtt{m})[^{\mathtt{m}'}/_{\mathtt{m}}])\ \rangle$, if $\mathtt{m}' \notin mN([\![\mathtt{TE}]\!])$ is fresh; (ii) $\langle\ \overline{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m})\ \rangle$, otherwise. Note that case (ii) can happen only if `m` and the occurrence of $\mathtt{m}'$ already in `TE` have the same signature, and it is not the case that both `m` and $\mathtt{m}'$ are provided methods (they can be both required or one of them required); otherwise, $\mathtt{TE}\,[\mathtt{rename\ m\ to\ m}']$ would have not been well-typed, which contradicts the hypothesis. In both cases, the result can be proved straightforwardly by induction on typing derivations.

Case $[\![\mathtt{TE}\,[\mathtt{rename\ f\ to\ f}']]\!]$**.** By induction, we have that $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}]\!] : \overline{\mu}$. Then we have two possible cases for the structural type of `this` to be used in the typing of the sequence of methods $[\![\mathtt{TE}]\!][^{\mathtt{f}'}/_{\mathtt{f}}]$: (i) $\langle\ \mathit{exclude}(\overline{\mathtt{F}}, \mathtt{f}) \cup (\mathit{choose}(\overline{\mathtt{F}}, \mathtt{f})[^{\mathtt{f}'}/_{\mathtt{f}}])\ \mid\ \overline{\sigma}\ \rangle$, if $\mathtt{f}' \notin fN([\![\mathtt{TE}]\!])$ is fresh; (ii) $\langle\ \mathit{exclude}(\overline{\mathtt{F}}, \mathtt{f})\ \mid\ \overline{\sigma}\ \rangle$, otherwise. Note that case (ii) can happen only if `f` and the occurrence of $\mathtt{f}'$ already in `TE` have the same type; otherwise, $\mathtt{TE}\,[\mathtt{rename\ f\ to\ f}']$ would have not been well-typed, which contradicts the hypothesis. In both cases, the result can be proved straightforwardly by induction on typing derivations. □

Lemma B.4. *If* $\mathtt{this} : \langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle,\ \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash_{\mathtt{co}} \mathtt{e} : \mathtt{R}\ \mid\ \langle\ \overline{\mathtt{F}}'\ \mid\ \overline{\sigma}'\ \rangle\ \mid\ \Phi$ *holds with respect to* P*, then it holds with respect to* $[\![\mathtt{P}]\!]$.

Proof. Let $\mathtt{P} = \overline{\mathtt{ID}}\ \overline{\mathtt{TD}}\ \overline{\mathtt{CD}}\ \mathtt{e}$. Then $[\![\mathtt{P}]\!] = \overline{\mathtt{ID}}\ [\![\overline{\mathtt{CD}}]\!]\ \mathtt{e}$. The result is straightforward, since the constraint-based typing rules in Fig. 21 do not use the trait table `TD`, and the only rule that uses the class table `CD`, rule (CT-New), does not distinguish between `CD` and $[\![\overline{\mathtt{CD}}]\!]$. □

Lemma B.5. *If* $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash_{\mathtt{co}} \mathtt{M} : \mu$ *holds with respect to* P*, then it holds with respect to* $[\![\mathtt{P}]\!]$.

Proof. Straightforward, since, by Lemma B.4, rule (CT-Class) in Fig. 19 does not use the trait table `TD` and does not distinguish between `CD` and $[\![\overline{\mathtt{CD}}]\!]$. □

Lemma B.6. *If* $\vdash_{\mathtt{co}} \mathtt{CD}$ *holds with respect to* P*, then* $\vdash_{\mathtt{co}} [\![\mathtt{CD}]\!]$ *holds with respect to* $[\![\mathtt{P}]\!]$.

Proof. Let $\mathtt{CD} = \ldots \mathtt{class\ C}\langle\overline{\alpha}\rangle\ \mathtt{implements\ I}\langle\overline{\mathtt{d}}\rangle\ \mathtt{by\ TE}\ \{\ \overline{\mathtt{F}};\ \}$, then

$$[\![\mathtt{CD}]\!] = \ldots \mathtt{class\ C}\langle\overline{\alpha}\rangle\ \mathtt{implements\ I}\langle\overline{\mathtt{d}}\rangle\ \mathtt{by}\{\ \overline{\mathtt{F}};\ \bullet;\ [\![\mathtt{TE}]\!]\ \}\ \{\ \overline{\mathtt{F}};\ \}$$

According to rule (CT-Class) in Fig. 19, $\vdash_{\mathtt{co}} \mathtt{CD}$ implies $\alpha_1 \vdash_{\mathtt{co}} \mathtt{TE} : \mu_1 \ldots \mu_p$, where

$$\forall i \in 1..p, \qquad \mu_i = \zeta_i\ \mid\ \langle\ \overline{\mathtt{F}}^{(i)}\ \mid\ \overline{\sigma}^{(i)}\ \rangle\ \mid\ (\overline{\mathtt{x}}^{(i)}, \Phi_i) \qquad \zeta_i = \mathtt{N}_i\ (\overline{\mathtt{N}}^{(i)}) \qquad \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle,\ \overline{\mathtt{x}}^{(i)} : \overline{\mathtt{N}}^{(i)} \vdash_{\mathtt{co}} \Phi_i[^{\mathtt{C}\langle\overline{\alpha}\rangle}/_{\chi}]$$

and $\cup_{i\in 1..p}\overline{\sigma}^{(i)} \subseteq \zeta_1 \ldots \zeta_p$.

By Lemma B.5, we have that $\alpha_1 \vdash_{\mathtt{co}} \mathtt{TE} : \mu_1 \ldots \mu_p$ holds also with respect to $[\![\mathtt{P}]\!]$.

By Lemma B.3, we have $\langle\ \overline{\mathtt{F}}\ \mid\ \overline{\zeta}\ \rangle \vdash_{\mathtt{co}} [\![\mathtt{TE}]\!] : \mu_1 \ldots \mu_p$, where $\overline{\mathtt{F}} = \overline{\mathtt{F}}^{(1)} \cup \ldots \cup \overline{\mathtt{F}}^{(p)}$ and $\overline{\zeta} = \zeta_1 \ldots \zeta_p$. Therefore both $\alpha_1 \vdash_{\mathtt{co}} \{\ \overline{\mathtt{F}};\ \bullet;\ [\![\mathtt{TE}]\!]\ \} : \mu_1 \ldots \mu_p$ and $\vdash_{\mathtt{co}} [\![\mathtt{CD}]\!]$ hold with respect to $[\![\mathtt{P}]\!]$. □

Proof of Theorem 6.2 (*Flattening Preserves* $\vdash_{\mathtt{co}}$*-typing*)**.** Straightforward, by Lemmas B.4 and B.6, according to rule (CT-Program) in Fig. 19. □

Appendix C. Proof of Theorem 8.2

First, we prove several auxiliary lemmas.

The Substitution Lemma is the core lemma of the soundness proof. It states how substituting values for variables and static domain owners affects the typing of expressions.

Lemma C.1 (*Substitution Type*)**.** *Let* $s = [\mathtt{v}'/\mathtt{box}, \mathtt{v}/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}, \overline{\mathtt{d}}/\overline{\alpha}]$*. If* $\vdash_{\mathtt{ox}} \mathtt{P}$*,* $\Theta(\mathtt{v}) = \mathtt{C}\langle\overline{\mathtt{d}}\rangle$*,* $\mathtt{v}' = \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v})$*,* $\Theta \vdash_{\mathtt{ox}} \mathtt{H}, \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle; \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \mathtt{L}$*, and* $\forall \mathtt{v}_k \in \overline{\mathtt{v}}$ *with* $\mathtt{v}_k \neq \mathtt{null} : \Theta(\mathtt{v}_k) <: \mathtt{G}_{\mathtt{x}_k}$*s, then* $\Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{L}s$.

Proof. Let $\mathtt{L} = \mathtt{U}\langle\overline{\mathtt{d}}_U\rangle$. By (V-TYPE), we get

$$\frac{\mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle;\ \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_\mathtt{o} \overline{\mathtt{d}}_U \qquad \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle;\ \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_\mathtt{o} \mathtt{box.c} \to \mathtt{d}_1^U \qquad |\mathit{params}(\mathtt{U})| = |\overline{\mathtt{d}}_U|}{\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle;\ \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_{\mathtt{ox}} \mathtt{U}\langle\overline{\mathtt{d}}_U\rangle}$$

Rule (V-TYPE-OID) gives us the following goals to show.

$$\Theta \vdash_{\mathtt{or}} \overline{\mathtt{d}}_U\ s\ (\mathrm{G1}) \qquad \Theta \vdash_{\mathtt{or}} \mathtt{d}_1^U\ s \to_\mathtt{r} \overline{\mathtt{d}}_U\ s\ (\mathrm{G2}) \qquad \Theta \vdash_{\mathtt{or}} \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v}).\mathtt{c} \to_\mathtt{r} \mathtt{d}_1^U\ s\ (\mathrm{G3}) \qquad |\mathit{params}(\mathtt{U})| = |\overline{\mathtt{d}}_U|\ (\mathrm{G4})$$

(G1) and (G4) follow directly.
For (G2), we do a induction on the accessibility relation.
By $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, we know that $\Theta;\ \mathtt{v};\ \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{C}\langle\overline{\mathtt{d}}\rangle$; thus $\Theta \vdash_{\mathtt{or}} \mathtt{d}_1 \to_\mathtt{r} \overline{\mathtt{d}}$, $\Theta \vdash_{\mathtt{or}} \mathtt{v}'.c \to_\mathtt{r} \mathtt{d}_1$. Let $\mathtt{G}_x = \mathtt{U}\langle\overline{\mathtt{d}}_x\rangle$.

Case (A-REFL), (A-OWNER), (A-GLOBAL). Immediate.
Case (A-PARAM). Thus $\mathtt{d}_1^U = \mathtt{box}.c$, $\mathtt{d}_2^U \in \overline{\alpha}$. $\mathtt{d}_1^U\ s = \mathtt{v}'.c = \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v}).c$. $\mathtt{d}_2^U\ s = \mathtt{d}_2 \in \overline{\mathtt{d}}$. Thus the preconditions of (A-PARAM) hold. If $\mathit{isBoxType}(\mathtt{C})$, then the preconditions of (A-PARAM) hold with $\iota' = \mathtt{v}$, and if $\neg\mathit{isBoxType}(\mathtt{C})$, the preconditions hold with $\iota' = \mathtt{v}'$ Thus (G2) holds in both cases.
Case (A-PARAM2). Thus $\mathtt{d}_1^U = \alpha_1, \mathtt{d}_2^U \in \overline{\alpha}$. Then $\mathtt{d}_1^U\ s = \mathtt{d}_1, \mathtt{d}_2^U\ s \in \overline{\mathtt{d}}$. (G2) follows from $\Theta \vdash_{\mathtt{or}} \mathtt{d}_1 \to_\mathtt{r} \overline{\mathtt{d}}$.
Case (A-PARAM-3). Thus $\mathtt{d}_1^U = \alpha_1$ and $\neg\mathit{isBoxType}(\mathtt{C})$. Then $\mathtt{d}_1^U\ s = \mathtt{d}_1$. We need to show that $\Theta \vdash_{\mathtt{or}} \mathtt{d}_1 \to_\mathtt{r} \mathtt{v}'.\mathtt{local}$. As $\neg\mathit{isBoxType}(\mathtt{C})$, $\mathtt{d}_1 = \mathtt{v}'.c$. Therefore (G2) holds by (AR-BOX).
Case (A-PARAM-4). Thus $\mathtt{d}_1^U = \alpha_1$. Then $\mathtt{d}_1^U\ s = \mathtt{d}_1$. We have to show that $\Theta \vdash_{\mathtt{or}} \mathtt{d}_1 \to_\mathtt{r} \mathtt{v}'.\mathtt{boundary}$. If $\neg\mathit{isBoxType}(\mathtt{C})$, then $\mathtt{d}_1 = \mathtt{v}'.c$. Then (G2) holds by (AR-BOX). If $\mathit{isBoxType}(\mathtt{C})$, then $\mathtt{v}' = \mathtt{v}$; therefore the precondition of (A-PARAM) holds. Thus (G2) holds.
Case (A-BOUNDARY). If $\mathtt{x} \neq \mathtt{this}$, we have to show that $\Theta \vdash_{\mathtt{or}} \mathtt{d}\ s \to_\mathtt{r} \mathtt{v}_x.\mathtt{boundary}$. Applying the induction hypotheses, we get $\Theta \vdash_{\mathtt{or}} \mathtt{d}\ s \to_\mathtt{r} \mathtt{d}_1^x\ s$. We know by Lemma C.3 that $\mathit{odom}(\Theta(\mathtt{v}_x)) = \mathit{odom}(\mathtt{G}_x\ s) = \mathtt{d}_x^1\ s$. Thus (G2) holds. If $\mathtt{x} = \mathtt{this}$, we know that $\mathit{isBoxType}(\mathtt{this})$ holds and therefore also $\mathit{isBoxType}(\Theta(\mathtt{v}))$. We have to show that $\Theta \vdash_{\mathtt{or}} \mathtt{d}\ s \to_\mathtt{r} \mathtt{v}.\mathtt{boundary}$. Applying the induction hypotheses we get $\Theta \vdash_{\mathtt{or}} \mathtt{d}\ s \to_\mathtt{r} \mathtt{d}_1\ s$. As $\mathtt{d}_1$ is a runtime domain, we can conclude that $\mathtt{d}_1\ s = \mathtt{d}_1 = \mathit{odom}(\Theta(\mathtt{v}))$; thus (G2) holds by (AR-BOUNDARY).
Case (A-BOUNDARY-2). If $\mathtt{x} \neq \mathtt{this}$, we have to show that $\Theta \vdash_{\mathtt{or}} \mathtt{v}_x.\mathtt{boundary} \to_\mathtt{r} \mathtt{d}_1^x\ s$. By $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, we know that $\Theta;\ \mathtt{v}_x;\ \mathtt{this} : \Theta(\mathtt{v}_x) \vdash_{\mathtt{ox}} \Theta(\mathtt{v}_x)$. Therefore we know that $\mathit{isBoxType}(\mathtt{G}_x)$. With Lemma C.3 we can conclude that $\Theta \vdash_{\mathtt{or}} \mathit{boxOwner}(\Theta(\mathtt{v}_x), \mathtt{v}_x).c \to_\mathtt{r} \mathtt{d}_1^x\ s$ and $\mathit{boxOwner}(\Theta(\mathtt{v}_x), \mathtt{v}_x) = \mathtt{v}_x$; thus (G2) holds. For $\mathtt{x} = \mathtt{this}$, we have to show that $\Theta \vdash_{\mathtt{or}} \mathtt{v}.\mathtt{boundary} \to_\mathtt{r} \mathtt{d}_1\ s$. Because $\mathit{isBoxType}(\Theta(\mathtt{v}))$ and thus $\mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v}) = \mathtt{v}$ hold, this can be concluded by (AR-PARAM).

(G3) can be shown by an induction on the accessibility relation, using the same arguments as above. □

Lemma C.2 (*Substitution*). *Let* $s = [\mathtt{v}'/\mathtt{box}, \mathtt{v}/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}, \overline{\mathtt{d}}/\overline{\alpha}]$. *If* $\vdash_{\mathtt{ox}} \mathtt{P}$, $\Theta(\mathtt{v}) = \mathtt{C}\langle\overline{\mathtt{d}}\rangle$, $\mathtt{v}' = \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v})$, $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, $\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle;\ \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{L}$, *and* $\forall \mathtt{v}_k \in \overline{\mathtt{v}}$ *with* $\mathtt{v}_k \neq \mathtt{null} : \Theta(\mathtt{v}_k) <: \mathtt{G}_{\mathtt{x}_k}\ s$, *then* $\Theta;\ \mathtt{v};\ \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{e}\ s : \mathtt{L}\ s$.

Proof. The proof is done by induction on the typing relation.

Case (T-NULL). Immediate.
Case (T-VAR). Let $\mathtt{e} = \mathtt{x}$. For $\mathtt{v}_x = \mathtt{null}$, the lemma follows directly; thus assume that $\mathtt{v}_x \neq \mathtt{null}$. We know that $\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \mathtt{x} : \mathtt{U}_x\langle\overline{\mathtt{d}}_x\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_{\mathtt{ox}} \mathtt{x} : \mathtt{U}_x\langle\overline{\mathtt{d}}_x\rangle$. By (V-ENV-VAR), we get $\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \mathtt{x} : \mathtt{U}_x\langle\overline{\mathtt{d}}_x\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_\mathtt{o} \mathtt{U}_x\langle\overline{\mathtt{d}}_x\rangle$.
Lemma C.1 then gives us $\Theta;\ \mathtt{v};\ \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{U}_x\langle\overline{\mathtt{d}}_x\rangle\ s$, which is the precondition for (V-OID). Thus we can conclude that $\Theta;\ \mathtt{v};\ \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{v}_x : \mathtt{U}_x\langle\overline{\mathtt{d}}_x\rangle\ s$.
Case (T-FIELD). Let $\mathtt{e} = \mathtt{e}.\mathtt{f}$. As field access is only allowed on this, we have $\mathtt{e} = \mathtt{this}$. Hence $\mathtt{e} = \mathtt{this.f}$. By the assumption $\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle;\ \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_{\mathtt{ox}} \mathtt{e.f} : \mathtt{L}$ and (T-FIELD), we get

$$\frac{\mathtt{N_f} = \mathit{field}(\mathtt{C}\langle\overline{\alpha}\rangle, \mathtt{f}) \qquad \begin{array}{c}\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_{\mathtt{ox}} \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle \\ \mathtt{C}\langle\overline{\alpha}\rangle;\ \mathtt{C}\langle\overline{\alpha}\rangle;\ \mathtt{this} \vdash_\mathtt{o} \mathtt{N_f} \triangleright \mathtt{L}\end{array} \qquad \varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_\mathtt{o} \mathtt{L}}{\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_\mathtt{x}} \vdash_{\mathtt{ox}} \mathtt{this.f} : \mathtt{L}}$$

By the definition of *field*, we get $\mathtt{N_f} = \mathtt{N}$, where N is the declared type of field f.
Let $\mathtt{N}'_\mathtt{f} = \mathit{field}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle, \mathtt{f})$. By the definition of *field*, we get that $\mathtt{N}'_\mathtt{f} = \mathtt{N_f}[\overline{\mathtt{d}}/\overline{\alpha}]$, because of the following argumentation.
Let $\mathtt{L}'$ be the type obtained by the following viewpoint adaptation, $\mathtt{C}\langle\overline{\mathtt{d}}\rangle;\ \mathtt{C}\langle\overline{\mathtt{d}}\rangle;\ \mathtt{v} \vdash_\mathtt{o} \mathtt{N}'_\mathtt{f} \triangleright \mathtt{L}'$. Applying the definition of viewpoint adaptation results in $\mathtt{L}' = \mathtt{N}'_\mathtt{f}[\mathit{obox}(\mathtt{d}_1, \mathtt{C}\langle\overline{\mathtt{d}}\rangle, \mathtt{v})/\mathtt{box}]$. As $\mathtt{d}_1$ can never be α, we obtain from the definition of *obox* that $\mathtt{L}' = \mathtt{N_f}[\mathit{boxOwner}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle, \mathtt{v})/\mathtt{box}, \overline{\mathtt{d}}/\overline{\alpha}]$. By (T-CLASS), we have that there is no x with $\mathtt{x} \in \mathtt{N_f}$, except for $\mathtt{x} = \mathtt{this}$. Hence we can conclude that $\varnothing;\ \mathtt{null};\ \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle \vdash_\mathtt{o} \mathtt{L}$. By Lemma C.1, we obtain $\Theta;\ \mathtt{v};\ \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{L}''$, where $\mathtt{L}'' = \mathtt{L}'[\mathtt{v}/\mathtt{this}]$. By exploiting the knowledge of possible owners in $\mathtt{N_f}$, we can conclude that $\mathtt{L}'' = \mathtt{L}\ s$.

Case (T-FIELD-UP). By applying the induction hypotheses twice and Lemma C.5.

Case (T-INVK). Let $\mathtt{e} = \mathtt{e.m}(\overline{\mathtt{e}})$. We need to show that

$$\frac{\begin{array}{c}\Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{e}\ s : \mathtt{G}'_{\mathtt{e}}\ (1) \qquad \Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \overline{\mathtt{e}\ s} : \overline{\mathtt{L}}'\ (2) \\ \mathit{mSig}(\mathtt{G}'_{\mathtt{e}}) = \ldots\ \mathtt{G}'_{\mathtt{m}}\ \mathtt{m}(\overline{\mathtt{G}'}_{\mathtt{m}})\ \ldots \qquad \mathtt{C}\langle\overline{\mathtt{d}}\rangle; \mathtt{G}'_{\mathtt{e}}; \mathtt{e}\ s \vdash_{\mathtt{ox}} \mathtt{G}'_{\mathtt{m}} \triangleright \mathtt{L}\ s\ (3) \qquad \mathtt{C}\langle\overline{\mathtt{d}}\rangle; \mathtt{G}'_{\mathtt{e}}; \mathtt{e}\ s \vdash_{\mathtt{ox}} \overline{\mathtt{G}'}_{\mathtt{m}} \triangleright \overline{\mathtt{G}''}_{\mathtt{m}}\ (4) \\ \Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{L}\ s\ (5) \qquad \Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \overline{\mathtt{G}''}_{\mathtt{m}}\ (6) \qquad \overline{\mathtt{L}}' <: \overline{\mathtt{G}''}_{\mathtt{m}}\ (7)\end{array}}{\Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{e}\ s.\mathtt{m}(\overline{\mathtt{e}\ s}) : \mathtt{L}\ s}$$

By the assumption, we have

$$\frac{\begin{array}{c}\varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{G}_{\mathtt{e}} \qquad \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \overline{\mathtt{e}} : \overline{\mathtt{L}} \\ \mathit{mSig}(\mathtt{G}_{\mathtt{e}}) = \ldots\ \mathtt{G}_{\mathtt{m}}\ \mathtt{m}(\overline{\mathtt{G}}_{\mathtt{m}})\ \ldots \qquad \mathtt{C}\langle\overline{\alpha}\rangle; \mathtt{G}_{\mathtt{e}}; \mathtt{e} \vdash_{\mathtt{ox}} \mathtt{G}_{\mathtt{m}} \triangleright \mathtt{L} \qquad \mathtt{C}\langle\overline{\alpha}\rangle; \mathtt{G}_{\mathtt{e}}; \mathtt{e} \vdash_{\mathtt{ox}} \overline{\mathtt{G}}_{\mathtt{m}} \triangleright \overline{\mathtt{G}}'_{\mathtt{m}} \\ \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \mathtt{L} \qquad \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \overline{\mathtt{G}}'_{\mathtt{m}} \qquad \overline{\mathtt{L}} <: \overline{\mathtt{G}}'_{\mathtt{m}}\end{array}}{\varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{L}}$$

By applying the induction hypotheses, we directly get (1) and (2), with $\mathtt{G}'_{\mathtt{e}} = \mathtt{G}_{\mathtt{e}}\ s$, $\overline{\mathtt{L}}' = \overline{\mathtt{L}}\ s$. Goal (5) follows by Lemma C.1.

To show (3), we make a case distinction on $\mathit{isBoxType}(\mathtt{G}_{\mathtt{e}})$. Let $\mathtt{N}_m\mathtt{m}(\overline{\mathtt{N}_m\mathtt{x}})$ be the declaration of method $\mathtt{m}$ in class $\mathtt{U}\langle\overline{\alpha}_{\mathtt{e}}\rangle$ and $\mathtt{G}_{\mathtt{e}} = \mathtt{U}\langle\overline{\mathtt{d}}_{\mathtt{e}}\rangle$. By (T-CLASS), and because we do not allow `this` as an owner in the surface syntax, we know that $\mathtt{N}_m$ and $\overline{\mathtt{N}_m}$ only contain $\overline{\alpha}_u$ and `box` as owners.

Case $\mathit{isBoxType}(\mathtt{G}_{\mathtt{e}})$. By $\mathtt{C}\langle\overline{\alpha}\rangle; \mathtt{G}_{\mathtt{e}}; \mathtt{e} \vdash_{\mathtt{ox}} \mathtt{G}_{\mathtt{m}} \triangleright \mathtt{L}$, we get $\mathit{validOwner}(\mathtt{e})$ and $\mathtt{L} = \mathtt{N}_m[\mathtt{e}/\mathtt{box}][\overline{\mathtt{d}}_e/\overline{\alpha}_e]$. As $\mathit{validOwner}(\mathtt{e})$ holds, we have $\mathtt{e} = \mathtt{x}$, $\mathtt{e} = \mathtt{this}$ or $\mathtt{e} = \mathtt{null}$. The last case cannot occur, as we know by (T-INVK) that e has a nominal type.

$\mathtt{e} = \mathtt{x}$. Applying s and simplifying the result with the knowledge about the possible owners in $\mathtt{N}_m$, we get $\mathtt{L}\ s = \mathtt{N}_m[\mathtt{v}_x/\mathtt{box}][\overline{\mathtt{d}}_e\ s/\overline{\alpha}_e]$. Let $\mathtt{L}'$ be the type resulting from the viewpoint adaptation $\mathtt{C}\langle\overline{\mathtt{d}}\rangle; \mathtt{G}'_{\mathtt{e}}; \mathtt{e}\ s \vdash_{\mathtt{ox}} \mathtt{G}'_{\mathtt{m}} \triangleright \mathtt{L}'$. Thus $\mathtt{L}' = \mathtt{N}_m[\mathtt{v}'/\mathtt{box}][\overline{\mathtt{d}}_e\ s/\overline{\alpha}_e]$. Therefore (3) holds.

$\mathtt{e} = \mathtt{this}$. Similar to above by exploiting $\mathtt{v}' = \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v})$.

Case $\neg\mathit{isBoxType}(\mathtt{G}_{\mathtt{e}})$. By $\mathtt{C}\langle\overline{\alpha}\rangle; \mathtt{G}_{\mathtt{e}}; \mathtt{e} \vdash_{\mathtt{ox}} \mathtt{G}_{\mathtt{m}} \triangleright \mathtt{L}$, we get $\mathtt{d}^1_{\mathtt{e}} = \alpha_1$ and $\mathtt{L} = \mathtt{N}_m[\overline{\mathtt{d}}_e/\overline{\alpha}_e]$. By the definition of $\mathit{boxOwner}$, we can conclude that $\mathit{boxOwner}(\mathtt{G}_{\mathtt{e}}\ s, \mathtt{e}\ s) = \mathit{boxOwner}(\mathtt{C}\langle\overline{\mathtt{d}}\rangle, \mathtt{v}) = \mathtt{v}$. Now it is easy to see that (3) holds.

For (4) and (6) are shown analogously to (3) and (5). (7) follows by Lemma C.5.

Case (T-NEW-CLASS). Let $\mathtt{e} = \mathtt{new}\ \mathtt{C}'\langle\overline{\mathtt{d}}'\rangle$. We have to show that

$$\frac{\ldots\ \mathtt{class}\ \mathtt{C}'\langle\overline{\alpha'}\rangle\ \mathtt{implements}\ \mathtt{N}\ \ldots \qquad \Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{C}'\langle\overline{\mathtt{d}}'\ s\rangle \qquad \mathtt{N}[\overline{\mathtt{d}}'\ s/\overline{\alpha'}] = \mathtt{L}\ s}{\Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{new}\ \mathtt{C}'\langle\overline{\mathtt{d}}'\ s\rangle : \mathtt{L}\ s}$$

We know that

$$\frac{\ldots\ \mathtt{class}\ \mathtt{C}'\langle\overline{\alpha'}\rangle\ \mathtt{implements}\ \mathtt{N}\ \ldots \qquad \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{o}} \mathtt{C}'\langle\overline{\mathtt{d}}'\rangle \qquad \mathtt{N}[\overline{\mathtt{d}}'/\overline{\alpha'}] = \mathtt{L}}{\varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{G}_{\mathtt{x}}} \vdash_{\mathtt{ox}} \mathtt{new}\ \mathtt{C}'\langle\overline{\mathtt{d}}'\rangle : \mathtt{L}}$$

By Lemma C.1, we know that $\Theta; \mathtt{v}; \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{d}}\rangle \vdash_{\mathtt{ox}} \mathtt{C}'\langle\overline{\mathtt{d}}'\ s\rangle$ holds.

By $\vdash_{\mathtt{o}} \mathtt{P}$, we know that the domains of N only contain `box` and α as owners. Therefore, we can conclude that $\mathtt{N}[\overline{\mathtt{d}}'\ s/\overline{\alpha'}] = \mathtt{N}s[\overline{\mathtt{d}}'/\overline{\alpha'}] = \mathtt{L}\ s$.

Case (T-CAST). Follows directly by applying the induction hypotheses and Lemma C.5. □

Lemma C.3. *If* $\mathtt{G} <: \mathtt{G}'$, *then* $\mathit{odom}(\mathtt{G}) = \mathit{odom}(\mathtt{G}')$.

Proof. This follows mainly by the precondition of rules (T-INTERFACE) and (T-CLASS), which states that $\mathtt{d}_1 = \alpha_1$. This ensures that subtyping has no influence on the owner domain. □

The next lemma states that, if a type is valid in certain context, a subtype of that type is also valid in that context.

Lemma C.4 (*Subtyping*). *Given* $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{L}$, $\Theta; \mathtt{v}'; \Gamma' \vdash_{\mathtt{ox}} \mathtt{L}'$, *and* $\mathtt{L} <: \mathtt{L}'$, *then it holds that* $\Theta; \mathtt{v}'; \Gamma' \vdash_{\mathtt{ox}} \mathtt{L}$.

Proof. We assume that $\mathtt{L} \neq \bot$; otherwise, the proof is immediate. Thus $\mathtt{L} = \mathtt{U}\langle\overline{\mathtt{d}}\rangle$, for some $\mathtt{U}$, $\overline{\mathtt{d}}$. As $\mathtt{L} <: \mathtt{L}'$, $\mathtt{L}' = \mathtt{U}'\langle\overline{\mathtt{d}}'\rangle$, for some $\mathtt{U}'$, $\overline{\mathtt{d}}'$. By the assumption that $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{L}$, we obtain that type L is a valid type under the given context, i.e., $\Theta \vdash_{\mathtt{or}} \overline{\mathtt{d}}$ (1), $\Theta \vdash_{\mathtt{or}} \mathtt{d}_1 \rightarrow_{\mathtt{r}} \overline{\mathtt{d}}$ (2), $\mathtt{v}_o = \mathit{boxOwner}(\Theta(\mathtt{v}), \mathtt{v})$, with $\Theta \vdash_{\mathtt{or}} \mathtt{v}_o.\mathtt{c} \rightarrow_{\mathtt{r}} \mathtt{d}_1$. By the second assumption, $\Theta; \mathtt{v}'; \Gamma' \vdash_{\mathtt{ox}} \mathtt{L}'$, we obtain $\mathtt{v}'_o = \mathit{boxOwner}(\Theta(\mathtt{v}'), \mathtt{v}')$, with $\Theta \vdash_{\mathtt{or}} \mathtt{v}'_o.\mathtt{c} \rightarrow_{\mathtt{r}} \mathtt{d}'_1$. By $\mathtt{L} <: \mathtt{L}'$ and Lemma C.3, we have that $\mathtt{d}_1 = \mathtt{d}'_1$, and hence $\Theta \vdash_{\mathtt{or}} \mathtt{v}'_o.\mathtt{c} \rightarrow_{\mathtt{r}} \mathtt{d}_1$. Together with (1) and (2), we can conclude that $\Theta; \mathtt{v}'; \Gamma' \vdash_{\mathtt{ox}} \mathtt{L}$. □

Lemma C.5. *If* $\mathtt{G} <: \mathtt{G}'$ *and* $s = [\mathtt{v}'/\mathtt{box}, \mathtt{v}''/\mathtt{this}, \overline{\mathtt{v}'''}/\overline{\mathtt{x}}, \overline{\mathtt{v}''''.c}/\overline{\alpha}]$, *then* $\mathtt{G}\,s <: \mathtt{G}'\,s$.

Proof. Straightforward, by induction on the subtyping relation. □

Lemma C.6. *If* $\Theta \vdash_{\mathtt{or}} H$ *then* $\forall \iota \in dom(\mathtt{H}) : \Theta(\iota) = \mathtt{C}\langle\overline{\mathtt{d}}\rangle$, $d_i = \mathtt{v}.c$, *i.e., the objects on the heap are all typed with a runtime type.*

Proof. Directly by (T-OID). □

Lemma C.7. *Let* $\mathtt{e} = \mathtt{e}_\square[\mathtt{e}']$. *If* $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{L}$, *then* $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{e}' : \mathtt{L}'$, *for some* $\mathtt{L}'$.

Proof. This lemma can be straightforwardly shown by a case analysis on each type rule, which shows that every subexpression of a typable expression is always typed under the same context. □

Lemma C.8. *Let* $\mathtt{e} = \mathtt{e}_\square[\mathtt{e}']$. *If* $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{L}_\mathtt{e}$, $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{e}' : \mathtt{L}_{\mathtt{e}'}$, $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{v}' : \mathtt{L}_{\mathtt{v}'}$, *and* $\mathtt{L}_{\mathtt{v}'} <: \mathtt{L}_{\mathtt{e}'}$, *then* $\Theta; \mathtt{v}; \Gamma \vdash_{\mathtt{ox}} \mathtt{e}_\square[\mathtt{v}'] : \mathtt{L}$, *for some* $\mathtt{L}$, *and* $\mathtt{L} <: \mathtt{L}_\mathtt{e}$.

Proof. This lemma can be straightforwardly shown by a case analysis on the possible evaluation contexts and an analysis of the corresponding type rule, which shows that replacing a subexpression by a value that can be typed to a subtype can only result in a subtype of the overall expression. □

Proof of Theorem 8.2 (*Subject Reduction*)**.** The proof is by induction on the rules of reduction semantics. Note that, for $\mathtt{v}' = \mathtt{null}$, the only typable expressions are $\mathtt{new}\mathtt{C}\langle\overline{\mathtt{d}}\rangle$, $\mathtt{null}$, and $(\mathtt{N})\mathtt{e}$, so in all other cases we can assume that $\mathtt{v}' \neq \mathtt{null}$.

Case (R-FIELD-READ). Thus $\mathtt{e} = \iota.\mathtt{f}$. From (R-FIELD-READ), we directly get that $\mathtt{H} = \mathtt{H}'$, $\Theta(\iota) = \mathtt{G}_\mathtt{t}$, and $\iota = \mathtt{v}'$. Choose $\Theta' = \Theta$. So we have still to show that $\Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{G}_\mathtt{v}, \mathtt{G}_\mathtt{v} <: \mathtt{G}_\mathtt{e}$. If $\mathtt{v} = \mathtt{null}$, $\mathtt{null}$ has type $\bot$ in any context, and $\bot$ is a subtype of any other type. In the following, we have $\mathtt{v} = \iota_v$. Rule (R-FIELD-READ) tells us that $\iota_v = \mathtt{H}(\iota)(\mathtt{f})$. We now have to show that $\Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota_v : \mathtt{G}_v, \mathtt{G}_v <: \mathtt{G}_\mathtt{e}$. By the assumption that $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, we know that for ι the following holds (T-HEAP):

$$\frac{\mathtt{G}_f = \mathit{field}(\mathtt{G}_\mathtt{t}, \mathtt{f}) \qquad \mathtt{G}_\mathtt{t}; \mathtt{G}_\mathtt{t}; \iota \vdash_{\mathtt{o}} \mathtt{G}_f \triangleright \mathtt{G}''_f \qquad \Theta(\iota) = \langle \mathtt{G}_\mathtt{t}, \overline{f} \mapsto \overline{\mathtt{v}_f} \rangle \qquad \Theta; \iota; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v}_f : \mathtt{G}'_f \qquad \mathtt{G}'_f <: \mathtt{G}''_f}{\Theta \vdash_{\mathtt{or}} \mathtt{H}}$$

We can conclude that $\iota_v = \mathtt{v}_f$. Therefore we get $\Theta; \iota; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota_v : \mathtt{G}'_f$ and $\mathtt{G}'_f <: \mathtt{G}''_f$. We have to show that $\mathtt{G}'_f = \mathtt{G}_v$ and $\mathtt{G}''_f <: \mathtt{G}_e$. As Θ is a function, (T-OID) always returns the same type if an object identifier is typed twice with the same context; thus $\mathtt{G}'_f = \mathtt{G}_v$. From (T-FIELD) and the assumption that $\Theta; \iota; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota.\mathtt{f} : \mathtt{G}_\mathtt{e}$, we get

$$\frac{\Theta; \iota; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota : \mathtt{G}_\mathtt{t} \qquad \mathtt{N}_\mathtt{f} = \mathit{field}(\mathtt{G}_\mathtt{t}, \mathtt{f}) \qquad \mathtt{G}_\mathtt{t}; \mathtt{G}_\mathtt{t}; \iota \vdash_{\mathtt{o}} \mathtt{N}_\mathtt{f} \triangleright \mathtt{G}_e \qquad \Theta; \iota; \Gamma \vdash_{\mathtt{ox}} \mathtt{G}_e}{\Theta; \iota; \Gamma \vdash_{\mathtt{ox}} \iota.\mathtt{f} : \mathtt{G}_e}$$

Thus we have $\mathtt{G}_e = \mathtt{G}''_f$.

Case (R-CALL). Thus $\mathtt{e} = \iota.\mathtt{m}(\overline{\mathtt{v}})$. By $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, (T-HEAP), and the definition of the function *type*, we know that $\Theta(\iota) = \mathit{type}(\mathtt{H}(\iota))$. If the expression evaluates to $\mathtt{null}$, i.e., $\mathtt{v} = \mathtt{null}$, the theorem can be seen directly. Therefore we assume in the following that $\mathtt{v} \neq \mathtt{null}$. By (R-CALL), we get

$$\frac{\mathtt{G}_\iota = \Theta(\iota) \qquad \mathit{mbody}(\mathtt{G}_\iota, \mathtt{m}) = (\overline{\mathtt{x}}, \mathtt{e}_b) \qquad \iota'' = \mathit{boxOwner}(\mathtt{G}_\iota, \iota) \qquad \mathtt{e}'_b = \mathtt{e}_b[\iota''/\mathtt{box}, \iota/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}] \qquad \mathtt{H}; \mathtt{e}'_b \Downarrow_\iota \mathtt{H}'; \mathtt{v}}{\mathtt{H}; \iota.\mathtt{m}(\overline{\mathtt{v}}) \Downarrow_{\mathtt{v}'} \mathtt{H}'; \mathtt{v}}$$

With the assumption that $\Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota.\mathtt{m}(\overline{\mathtt{v}}) : \mathtt{G}_\mathtt{e}$ and (T-INVK), we get

$$\frac{\begin{array}{c}\Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota : \mathtt{G}_\iota \\ \Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \overline{\mathtt{v}} : \overline{\mathtt{L}} \qquad \mathit{mSig}(\mathtt{G}_\iota) = \ldots\ \mathtt{G}_\mathtt{m}\,\mathtt{m}(\overline{\mathtt{G}_\mathtt{m}})\ \ldots \qquad \mathtt{G}_\mathtt{t}; \mathtt{G}_\iota; \iota \vdash_{\mathtt{ox}} \mathtt{G}_\mathtt{m} \triangleright \mathtt{G}_\mathtt{e} \\ \mathtt{G}_\mathtt{t}; \mathtt{G}_\iota; \iota \vdash_{\mathtt{ox}} \overline{\mathtt{G}_\mathtt{m}} \triangleright \overline{\mathtt{G}'_\mathtt{m}} \qquad \Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{G}_\mathtt{e} \qquad \Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \overline{\mathtt{G}'_\mathtt{m}} \qquad \overline{\mathtt{L}} <: \overline{\mathtt{G}'_\mathtt{m}}\end{array}}{\Theta; \mathtt{v}'; \mathtt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{o}} \iota.\mathtt{m}(\overline{\mathtt{v}}) : \mathtt{G}_\mathtt{e}}$$

Let $\mathtt{G}_\iota = \mathtt{C}\langle\overline{\mathtt{d}}\rangle$. Note that Lemma C.6 guarantees that $\mathtt{G}_\iota$ is not an interface type and all domains are of the form $\mathtt{v}_d.c$. From $\vdash_{\mathtt{ox}} \mathtt{P}$, we know that (T-METHOD) holds for method $\mathtt{m}$ in class $\mathtt{C}\langle\overline{\alpha}\rangle$. Thus,

$$\frac{\text{(T-METHOD)}\quad \mathtt{S} = \mathtt{N}\ \mathtt{m}\ (\overline{\mathtt{N}}\ \overline{\mathtt{x}}) \qquad \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle \vdash_{\mathtt{ox}} \mathtt{N} \cup \overline{\mathtt{N}} \qquad \varnothing; \mathtt{null}; \mathtt{this} : \mathtt{C}\langle\overline{\alpha}\rangle, \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash_{\mathtt{ox}} \mathtt{e}_m : \mathtt{L} \qquad \mathtt{L} <: \mathtt{N}}{\mathtt{C}\langle\overline{\alpha}\rangle \vdash_{\mathtt{ox}} \mathtt{S}\{\mathtt{return}\ \mathtt{e}_m\}}$$

In order to apply the induction hypotheses on $\mathtt{H}; \mathtt{e}'_b \Downarrow_\iota \mathtt{H}'; \mathtt{v}$, we have to show the assumptions of the substitution theorem. Let $s = [\iota''/\mathtt{box}, \iota/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}, \overline{\mathtt{d}}/\overline{\alpha}]$. $\vdash_{\mathtt{ox}} \mathtt{P}$, $\Theta \vdash_{\mathtt{ox}} \mathtt{H}$, $\Theta(\iota) = \mathtt{G}_\iota$, and $\mathtt{this} \notin \mathtt{e}'_b$ are given by the

assumptions and (R-CALL). It remains to show that $\Theta; \iota; \texttt{this} : \mathtt{G}_\iota \vdash_{\mathtt{ox}} \mathtt{e}'_b : \mathtt{G}_\mathtt{e}$. We will do so by Lemma C.2. By (R-CALL), we know that $\Theta(\iota) = \mathtt{G}_\iota = \mathtt{C}\langle\overline{\mathtt{d}}\rangle$, $\iota'' = \mathit{boxOwner}(\Theta(\iota), \iota)$, and $\varnothing; \texttt{null}; \texttt{this} : \mathtt{C}\langle\overline{\alpha}\rangle; \overline{\mathtt{x}} : \overline{\mathtt{N}} \vdash_{\mathtt{ox}} \mathtt{e}_m : \mathtt{L}$ is given by (T-METHOD). By the definition of the function *mSig*, (T-METHOD), and (T-INVK), we get $\mathtt{G}_m = \mathtt{N}[\overline{\mathtt{d}}/\overline{\alpha}]$ and $\overline{\mathtt{G}_m} = \overline{\mathtt{N}}[\overline{\mathtt{d}}/\overline{\alpha}]$. By $\mathtt{G}_\mathtt{t}; \mathtt{G}_\iota; \iota \vdash_{\mathtt{ox}} \overline{\mathtt{G}_\mathtt{m}} \triangleright \overline{\mathtt{G}'_\mathtt{m}}$ of (T-INVK), we can conclude that $\overline{\mathtt{G}'_\mathtt{m}} = \overline{\mathtt{N}}[\iota''/\mathtt{box}, \overline{\mathtt{d}}/\overline{\alpha}]$, because $\mathtt{G}_t$ is a runtime type and *validOwner*(ι) holds. We know that $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \overline{\mathtt{v}} : \overline{\mathtt{L}}$; thus for every $\mathtt{v}_k \in \overline{\mathtt{v}}$ with $\mathtt{v}_k \neq \texttt{null}$ we have $\Theta(\mathtt{v}_k) <: \mathtt{N}_k[\iota''/\mathtt{box}, \overline{\mathtt{d}}/\overline{\alpha}]$. Applying s (Lemma C.5) yields $\Theta(\mathtt{v}_k) <: \mathtt{N}_l\, s$, as $\Theta(\mathtt{v}_k)$ is a runtime types and the substitution does not change anything. We can now apply Lemma C.2, and we get $\Theta; \iota; \texttt{this} : \mathtt{G}_\iota \vdash_{\mathtt{ox}} \mathtt{e}'_b : \mathtt{L}\, s$, which is the last assumption needed for the induction hypotheses (with $\mathtt{G}_e = \mathtt{L}\, s$). By the induction hypotheses, we can conclude that there exists a $\Theta'_{\mathtt{IH}}$ with $\Theta \subseteq \Theta_{\mathtt{IH}}, \Theta_{\mathtt{IH}} \vdash_{\mathtt{ox}} \mathtt{H}', \Theta_{\mathtt{IH}}; \iota; \texttt{this} : \mathtt{G}_\iota \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{G}'_\mathtt{v}$, and $\mathtt{G}'_\mathtt{v} <: \mathtt{G}_\mathtt{e} = \mathtt{L}\, s$. Choose $\Theta' = \Theta_{\mathtt{IH}}$. Thus $\mathtt{G}_{v'} = \mathtt{G}_v$. From (T-INVK), we get $\mathtt{G}_\mathtt{t}; \mathtt{G}_\iota; \iota \vdash_{\mathtt{ox}} \mathtt{N}[\overline{\mathtt{d}}/\overline{\alpha}] \triangleright \mathtt{G}_\mathtt{e}$, thus $\mathtt{G}_\mathtt{e} = \mathtt{N}[\iota''/\mathtt{box}, \overline{\mathtt{d}}/\overline{\alpha}]$. By (T-METHOD) and Lemma C.5 we know that $\mathtt{L}\, s <: \mathtt{N}\, s = \mathtt{N}[\iota''/\mathtt{box}, \iota/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}, \overline{\mathtt{d}}/\overline{\alpha}] = \mathtt{G}_\mathtt{e}[\iota/\mathtt{this}, \overline{\mathtt{v}}/\overline{\mathtt{x}}] = \mathtt{G}_\mathtt{e}$, because $\mathtt{G}_\mathtt{e}$ is a runtime type. In summary, we have $\mathtt{G}_\mathtt{v} <: \mathtt{G}_\mathtt{e}$. From $\mathtt{G}_\mathtt{v} <: \mathtt{G}_\mathtt{e}$, $\Theta'; \iota; \texttt{this} : \mathtt{G}_\iota \vdash_{\mathtt{ox}} \mathtt{G}_\mathtt{v}$ (induction hypotheses), and $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{G}_\mathtt{e}$ (assumption of the theorem), we get by Lemma C.4 that $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{G}_\mathtt{v}$. We can then conclude that $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{G}_\mathtt{v}$. We now have $\Theta \subseteq \Theta'$, $\Theta' \vdash_{\mathtt{ox}} \mathtt{H}', \Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{G}_\mathtt{v}$, and $\mathtt{G}_\mathtt{v} <: \mathtt{G}_\mathtt{e}$, i.e., all results of the theorem.

Case (R-FIELD-UPDATE). By the definition of that rule, we have $\mathtt{e} = \iota.\mathtt{f} = \mathtt{v}$, $\mathtt{o} = \mathtt{H}(\iota)[\mathtt{f} \mapsto \mathtt{v}]$, and $\mathtt{H}' = \mathtt{H}[\iota \mapsto \mathtt{o}]$. By applying (T-FIELD-UPDATE) and (T-FIELD) to the assumption $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota.\mathtt{f} : \mathtt{G}_\mathtt{e}$, we get $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{L}_\mathtt{v}$ (1), $\mathtt{L}_\mathtt{v} <: \mathtt{G}_\mathtt{e}$ (2), $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \iota : \mathtt{G}_\iota$, $\mathtt{G}_\mathtt{f} = \mathit{field}(\mathtt{G}_\iota, \mathtt{f})$, $\mathtt{G}_\mathtt{t}; \mathtt{G}_\iota; \iota \vdash_{\mathtt{o}} \mathtt{G}_\mathtt{f} \triangleright \mathtt{G}_\mathtt{e}$ (3), and $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{G}_\mathtt{e}$. By (1) and (2) we directly obtain two of the three goals to be shown. It remains to show that there exists a Θ' with $\Theta \subseteq \Theta'$ and $\Theta' \vdash_{\mathtt{or}} \mathtt{H}'$. Choose $\Theta' = \Theta$. To show that $\Theta \vdash_{\mathtt{or}} \mathtt{H}'$, we have to show that the new value v of field f of object ι is a valid value, i.e., $\Theta; \iota; \texttt{this} : \mathtt{G}_\iota \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{L}_\mathtt{v}, \mathtt{L}_\mathtt{v} <: \mathtt{G}'_\mathtt{e}$, and $\mathtt{G}_\iota; \mathtt{G}_\iota; \iota \vdash_{\mathtt{o}} \mathtt{G}_\mathtt{f} \triangleright \mathtt{G}'_\mathtt{e}$. But this follows from the fact that, by (R-FIELD-UPDATE), $\mathtt{v}' = \iota$, and the assumption that $\Theta(\mathtt{v}') = \mathtt{G}_\mathtt{t}$, which means that $\mathtt{G}_\mathtt{t} = \mathtt{G}_\iota$. Hence, by (3), $\mathtt{G}_\mathtt{e} = \mathtt{G}'_\mathtt{e}$, and we can conclude that $\Theta \vdash_{\mathtt{or}} \mathtt{H}'$.

Case (R-NEW-OBJECT). Thus $\mathtt{H}; \mathtt{C}\langle\overline{\mathtt{d}}\rangle \Downarrow_{\mathtt{v}'} \mathtt{H}'; \iota_\mathtt{C}$. Choose $\Theta' = [\iota_\mathtt{C} \to \mathtt{C}\langle\overline{\mathtt{d}}\rangle]\Theta$; then the conclusion follows directly.

Case (R-CAST-NULL). Clear.

Case (R-CAST). Straightforward.

Case (R-CONTEXT). By the definition of that rule, we have $e = \mathtt{e}_\square[\mathtt{e}']$ (1), $\mathtt{H}; \mathtt{e}' \Downarrow_{\mathtt{v}'} \mathtt{H}''; \mathtt{v}''$ (2), and $\mathtt{H}''; \mathtt{e}_\square[\mathtt{v}''] \Downarrow_{\mathtt{v}'} \mathtt{H}'; \mathtt{v}$ (3). By the assumption that $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{e} : \mathtt{G}_\mathtt{e}$ and Lemma C.7, we obtain that there exists a $\mathtt{L}_{\mathtt{e}'}$ with $\Theta; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{e}' : \mathtt{L}_{\mathtt{e}'}$ (4). We can now apply the induction hypothesis, and we get that there exists a $\Theta'' \subseteq \Theta$ with $\Theta''; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v}'' : \mathtt{L}_{\mathtt{v}''}, \mathtt{L}_{\mathtt{v}''} <: \mathtt{L}_{\mathtt{e}'}$, and $\Theta'' \vdash_{\mathtt{ox}} \mathtt{H}''$. By using Lemma C.8, we obtain that exists a type L with $\Theta''; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{e}_\square[\mathtt{v}''] : \mathtt{L}$, and $\mathtt{L} <: \mathtt{G}_\mathtt{e}$ (5). We can now apply the induction hypothesis on (3), and we get that there exists a $\Theta' \subseteq \Theta''$ with $\Theta'; \mathtt{v}'; \texttt{this} : \mathtt{G}_\mathtt{t} \vdash_{\mathtt{ox}} \mathtt{v} : \mathtt{G}_\mathtt{v}, \mathtt{G}_\mathtt{v} <: \mathtt{L}$ (6), and $\Theta' \vdash_{\mathtt{ox}} \mathtt{H}'$. From (5) and (6) and the transitivity rule of subtyping, $\mathtt{G}_\mathtt{v} <: \mathtt{G}_\mathtt{e}$, closing the case. □

References

[1] R. Agarwal, S. Stoller, Type inference for parameterized race-free Java, in: Verification, Model Checking, and Abstract Interpretation, Springer, 2003, pp. 77–108.
[2] J. Aldrich, Using types to enforce architectural structure, in: Seventh Working IEEE/IFIP Conference on Software Architecture, IEEE, 2008, pp. 211–220.
[3] J. Aldrich, C. Chambers, Ownership domains: Separating aliasing policy from mechanism, in: ECOOP, in: LNCS, vol. 3086, Springer, 2004, pp. 1–25.
[4] J. Aldrich, C. Chambers, D. Notkin, ArchJava: connecting software architecture to implementation, in: Software Engineering, 2002. ICSE 2002, IEEE, 2002, pp. 187–197.
[5] J. Aldrich, V. Kostadinov, C. Chambers, Alias annotations for program understanding, in: Proc. OOPSLA 2002, ACM Press, 2002, pp. 311–330.
[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, G.-W. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstad, The Fortress Language Specification, V. 1.0, 2008.
[7] D. Ancona, G. Lagorio, E. Zucca, Jam—designing a Java extension with mixins, ACM Trans. Program. Lang. Syst. 25 (5) (2003) 641–712.
[8] L. Bettini, F. Damiani, M.D. Luca, K. Geilmann, J. Schäfer, A calculus for boxes and traits in a Java-like setting, in: Coordination, in: LNCS, vol. 6116, Springer, 2010, pp. 46–60.
[9] L. Bettini, F. Damiani, I. Schaefer, Implementing software product lines using traits, in: SAC, ACM, 2010, pp. 2096–2102.
[10] L. Bettini, F. Damiani, I. Schaefer, F. Strocco, TraitRecordJ: a programming language with traits and records, Science of Computer Programming (2011). doi:10.1016/j.scico.2011.06.007.
[11] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Squeak by Example, Square Bracket Associates, 2007.
[12] V. Bono, F. Damiani, E. Giachino, Separating type, behavior, and state to achieve very fine-grained reuse, in: FTfJP, 2007. (www.cs.ru.nl/ftfjp/).
[13] V. Bono, F. Damiani, E. Giachino, On traits and types in a Java-like setting, in: TCS (Track B), in: IFIP, vol. 273, Springer, 2008, pp. 367–382.
[14] C. Boyapati, SafeJava: a unified type system for safe programming. Ph.D. thesis, Massachusetts Institute of Technology, February 2004.
[15] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: preventing data races and deadlocks, in: Proc. OOPSLA 2002, ACM Press, November 2002, pp. 211–230.
[16] C. Boyapati, B. Liskov, L. Shrira, Ownership types for object encapsulation, in: POPL, ACM Press, 2003, pp. 213–223.
[17] C. Boyapati, M. Rinard, A parameterized type system for race-free Java programs, in: Proc. OOPSLA 2001, ACM Press, October 2001, pp. 56–69.
[18] G. Bracha, W. Cook, Mixin-based inheritance, in: OOPSLA, in: SIGPLAN NOTICES, vol. 25(10), ACM, 1990, pp. 303–311.
[19] N. Cameron, S. Drossopoulou, J. Noble, M. Smith, Multiple ownership, in: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA '07, ACM, 2007, p. 460.
[20] N. Cameron, J. Noble, T. Wrigstad, Tribal ownership, in: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10, ACM, 2010, pp. 618–633.
[21] D. Clarke, Object ownership and containment. Ph.D. thesis, Univ. New South Wales, 2001.
[22] D. Clarke, S. Drossopoulou, Ownership, encapsulation, and the disjointness of type and effect, in: Proc. OOPSLA 2002, ACM Press, November 2002, pp. 292–310.

[23] D. Clarke, J. Noble, J.M. Potter, Simple ownership types for object containment, in: J.L. Knudsen (Ed.), Proc. ECOOP 2001, in: Lecture Notes in Computer Science, vol. 2072, Springer, June 2001, pp. 53–76.
[24] D. Clarke, J. Potter, J. Noble, Ownership types for flexible alias protection, in: OOPSLA, ACM Press, 1998, pp. 48–64.
[25] D. Clarke, T. Wrigstad, External uniqueness is unique enough, in: ECOOP 2003, in: Lecture Notes in Computer Science, vol. 2743, Springer, Berlin, Heidelberg, 2003.
[26] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, A.J. Summers, Universe types for topology and encapsulation, in: FMCO, in: LNCS, vol. 5382, Springer, 2008, pp. 72–112.
[27] W. Dietl, S. Drossopoulou, P. Müller, Generic universe types, in: ECOOP, in: LNCS, vol. 4609, Springer, 2007, pp. 28–53.
[28] W. Dietl, M. Ernst, P. Müller, Tunable static inference for generic universe types, in: European Conference on Object-Oriented Programming (ECOOP), 2011.
[29] W. Dietl, P. Müller, Universes: Lightweight ownership for JML, Journal of Object Technology 4 (8) (2005) 5–32.
[30] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: a mechanism for fine-grained reuse, ACM Trans. Program. Lang. Syst. 28 (2) (2006) 331–388.
[31] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical Computer Science 103 (2) (1992) 235–271.
[32] K. Fisher, J. Reppy, A typed calculus of traits, in: FOOL, 2004.
[33] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: POPL, ACM, 1998, pp. 171–183.
[34] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java™ Language Specification, 2nd ed., Addison-Wesley, June 2000.
[35] C. Grothoff, J. Palsberg, J. Vitek, Encapsulating objects with confined types, in: Proc. OOPSLA 2001, ACM Press, October 2001, pp. 241–253.
[36] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450.
[37] N. Krishnaswami, J. Aldrich, Permission-based ownership: Encapsulating state in higher-order typed languages, in: Proc. PLDI'05, ACM Press, 2005, pp. 96–106.
[38] T. Kühne, D. Schreiber, Can programming be liberated from the two-level style: multi-level programming with DeepJava, in: OOPSLA, ACM, 2007, pp. 229–244.
[39] G. Lagorio, M. Servetto, E. Zucca, Flattening versus direct semantics for Featherweight Jigsaw, in: FOOL, 2009. (www.cs.hmc.edu/~stone/FOOL/).
[40] K.R.M. Leino, P. Müller, Object invariants in dynamic contexts, in: ECOOP, in: LNCS, vol. 3086, Springer, 2004, pp. 491–516.
[41] M. Limberghen, T. Mens, Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems, Object Oriented Systems 3 (1) (1996) 1–30.
[42] L. Liquori, A. Spiwack, Extending feathertrait Java with interfaces, Theoretical Computer Science 398 (1-3) (2008) 243–260.
[43] L. Liquori, A. Spiwack, FeatherTrait: a modest extension of featherweight Java, ACM TOPLAS 30 (2) (2008) 1–32.
[44] Y. Liu, S. Smith, Pedigree types, in: Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, 2008.
[45] Y. Lu, J. Potter, On ownership and accessibility, in: ECOOP, in: LNCS, vol. 4067, Springer, 2006, pp. 99–123.
[46] Y. Lu, J. Potter, Protecting representation with effect encapsulation, in: In Proc. POPL '06, ACM Press, 2006, pp. 359–371.
[47] K. Ma, J. Foster, Inferring aliasing and encapsulation properties for Java, in: Proc. OOPSLA 2007, ACM, 2007, pp. 423–440.
[48] P. Müller, Modular Specification and Verification of Object-Oriented Programs, in: LNCS, vol. 2262, Springer, 2002.
[49] P. Müller, A. Poetzsch-Heffter, A type system for controlling representation exposure in Java, in: FTfJP, 2000. (www.cs.ru.nl/ftfjp/).
[50] P. Müller, A. Rudich, Ownership transfer in universe types, in: Proc. OOPSLA 2007, ACM, 2007, pp. 461–478.
[51] S. Nägeli, Ownership in design patterns. Master's thesis, March, 2006.
[52] O. Nierstrasz, S. Ducasse, N. Schärli, Flattening traits, JOT 5 (4) (2006) 129–148.
[53] J. Noble, J. Vitek, J. Potter, Flexible alias protection, in: E. Jul (Ed.), Proc. ECOOP' 98, in: LNCS, vol. 1445, Springer, 1998, pp. 158–185.
[54] M. Odersky, The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.
[55] A. Poetzsch-Heffter, K. Geilmann, J. Schäfer, Infering ownership types for encapsulated object-oriented program components, in: Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm, in: LNCS, vol. 4444, Springer, 2007.
[56] A. Poetzsch-Heffter, J. Schäfer, Modular specification of encapuslated object-oriented components, in: FMCO, in: LNCS, vol. 4111, Springer, 2006, pp. 313–341.
[57] A. Poetzsch-Heffter, J. Schäfer, A representation-independent behavioral semantics for object-oriented components, in: FMOODS, in: LNCS, vol. 4468, Springer, 2007, pp. 157–173.
[58] A. Potanin, J. Noble, D. Clarke, R. Biddle, Generic ownership for generic Java, in: OOPSLA, ACM Press, 2006, pp. 311–324.
[59] J. Reppy, A. Turon, A foundation for trait-based metaprogramming, in: FOOL/WOOD, 2006.
[60] J. Reppy, A. Turon, Metaprogramming with traits, in: ECOOP, in: LNCS, vol. 4609, Springer, 2007, pp. 373–398.
[61] J. Schäfer, A. Poetzsch-Heffter, A parameterized type system for simple loose ownership domains, Journal of Object Technology (JOT) 5 (6) (2007) 71–100.
[62] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: ECOOP, in: LNCS, vol. 2743, Springer, 2003, pp. 248–274.
[63] C. Smith, S. Drossopoulou, *Chai*: Traits for Java-like languages, in: ECOOP, in: LNCS, vol. 3586, Springer, 2005, pp. 453–478.
[64] M. Smith, Towards an effects system for ownership domains, in: ECOOP Workshop-FTfJP 2005, July 2005.
[65] C. Szyperski, D. Gruntz, S. Murer, Component Software — Beyond Object-Oriented Programming, 2nd ed., Addison-Wesley, 2002.
[66] D. Ungar, C. Chambers, B.-W. Chang, U. Hölzle, Organizing programs without classes, Lisp and Symbolic Computation 4 (3) (July 1991) 223–242.
[67] J. Vitek, B. Bokowski, Confined types in Java, Software – Practice and Experience 31 (6) (2001) 507–532.
[68] T. Zhao, J. Palsberg, J. Vitek, Lightweight confinement for featherweight Java, in: R. Crocker, G.L. S. Jr. (Eds.), Proc. OOPSLA 2003, ACM Press, October 2003, pp. 135–148.